DEVELOPING A SOPHISTICATED USER INTERFACE TO SUPPORT DOMAIN-ORIENTED APPLICATION COMPOSITION AND GENERATION SYSTEMS

THESIS

Jay A. Cossentine
Captain, USAF

AFIT/GCS/ENG/93D-04

AFIT/GCS/ENG/93D-04

Developing a Sophisticated User Interface to Support Domain-Oriented Application Composition and Generation Systems

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Jay A. Cossentine, B.S.E.E.

Captain, USAF

December 15, 1993

Acknowledgements

I wish to thank my research committee for their help and advice. Major David "Doctor Dave" Luginbuhl helped me see clearly on several occasions, and Dr. Tom Hartrum's comments and insight as reader were greatly appreciated. I wish to convey a sincere thanks to Major Paul Bailor, my thesis advisor. Although he earned the nickname of "Riddler," his questions and guidance were not only thought provoking but relevant and helpful. I'd also like to express my thanks to my fellow KBSE researchers for all the support and cooperation they gave me during our research. I wish to offer a special thanks to Alice Campbell for proof-reading this thesis and making many constructive comments.

For my sanity ar sense of humor, I'd like to thank John Keller, Warren Gool, Jeff Miller, and Joe Fullenkamp. Without their help I would surely have had less fun and more difficulty throughout this AFIT experience. Thanks to Vince Droddy for teaching me to relax over a game of cutthroat Pinochle.

I want to thank my parents and sister for their faith in me and their unswerving confidence and support. I want to thank my children, Betsy and Scott, for understanding that Daddy had to spend a lot of time at school, and loving me all the more. Lastly, and most especially, I need to thank my wife, Gloria. Her love, understanding, and support got me through this program. Her faith and encouragement kept me going through my difficulties and triumphs.

Jay A. Cossentine

Table of Contents

	Page
Acknowledgements	ii
List of Figures	viii
List of Tables	x
Abstract	xi
I. Introduction	1-1
1.1 Background	1-1
1.2 Problem	1-4
1.3 Assumptions	1-8
1.4 Scope	1-8
1.5 Sequence of Presentation	1-9
II. Literature Review	2-1
2.1 Introduction	2-1
2.2 Icon and Metaphor Design	2-1
2.2.1 Icons in Everyday Use	2-1
2.2.2 Icons in Computer Use	2-3
2.3 Window Design and Layout	2-7
2.4 Color Usage	2-8
2.4.1 Window Color	2-11
2.4.2 Icon Color	2-11
2.5 Lisp-based Graphics Software	2-13
2.5.1 INTERVISTA	2-13
2.5.2 Common Windows	2-13

Page

2.5.3 CLX 2-14

2.5.4 CLIM 2-14

2.5.5 GINA 2-14

2.5.6 CLUE 2-15

2.5.7 Garnet 2-15

2.5.8 LispView 2-15

2.5.9 Winterp 2-16

2.5.10 XIT 2-16

2.6 Conclusion 2-17

III. System Enhancements 3-1

3.1 Introduction 3-1

3.2 Operational Concept 3-3

3.2.1 Defining the Application 3-4

3.2.2 Building the Subsystems 3-5

3.2.3 Specifying the Update Algorithm 3-11

3.2.4 Connecting the Imports and Exports 3-14

3.2.5 Executing the Application 3-18

3.3 Testing 3-19

3.3.1 Functionality Testing 3-20

3.3.2 Usability Testing 3-21

3.3.3 Informal Testing 3-22

3.4 Conclusion 3-22

IV. Design and Implementation of AVSI II 4-1

4.1 Introduction 4-1

4.2 Design and Implementation Approach 4-2

4.3 Domain Specific Icons 4-3

Page

4.3.1 Metaphor Development 4-4

4.3.2 Icon Display 4-7

4.3.3 Refresh and Redraw 4-8

4.3.4 Icon Scalability 4-9

4.3.5 Required Attributes 4-10

4.4 Update Algorithm 4-11

4.4.1 Controllee Window 4-12

4.4.2 Update Object Creation 4-12

4.5 Import and Export Areas 4-13

4.5.1 Import - Export Icon Creation 4-13

4.5.2 Primitive Icon Connection 4-14

4.5.3 Inter-Subsystem Communication 4-16

4.5.4 AVSI Compatibility 4-16

4.6 Application Executive 4-17

4.6.1 Execution Mode Support 4-17

4.6.2 Event Queue Support 4-18

4.7 Miscellaneous 4-18

4.7.1 Color in AVSI II 4-18

4.7.2 Multiple Link Function 4-19

4.7.3 Parse and Save Functions 4-20

4.8 Summary 4-20

V. Testing and Validation of AVSI II 5-1

5.1 Validation Domain 5-1

5.2 Compatibility Testing 5-2

5.3 Usability Testing 5-3

5.3.1 User Workload 5-3

5.3.2 Workload Analysis 5-8

Page

5.3.3 User Evaluations 5-9

5.4 AVSI II's Shortcomings 5-11

5.5 Summary 5-13

VI. Conclusion and Recommendations 6-1

6.1 Results and Conclusions 6-2

6.1.1 Domain-Specific Graphics 6-2

6.1.2 Update Algorithm 6-3

6.1.3 Import-Export Areas 6-3

6.2 Recommendations for Further Research 6-4

6.3 Summary 6-8

Appendix A. Sample Session for AVSI II A-1

A.1 Starting AVSI A-1

A.2 Create a New Application A-3

A.3 Edit the Application A-3

A.3.1 To add a controlling subsystem-obj to the application A-3

A.3.2 To create the application-obj's update-algorithm . . A-5

A.4 Edit the Subsystems A-7

A.4.1 To add the subsystem A-8

A.4.2 To add the primitive objects A-10

A.4.3 To connect DRIVER's Imports and Exports A-12

A.4.4 To build DRIVER's Update Algorithm A-16

A.4.5 To connect BCD-EXCESS3's Imports and Exports . A-18

A.4.6 To build BCD-EXCESS3's Update Algorithm A-22

A.5 Perform Semantic Checks A-23

A.6 Execute the Application A-23

Page

Appendix B. Instructions for Developing a Metaphor Set B-1

B.1 Design Principles B-1

B.2 Bitmap Construction B-6

B.2.1 Available Tools B-6

B.2.2 Required Sizes B-6

B.3 File Creation B-8

B.3.1 <*bitmap*>-l B-8

B.3.2 <*bitmap*>-s B-8

B.3.3 var-<*domain*> B-8

B.3.4 vsl-<*domain*> B-9

B.4 Conclusion B-9

Appendix C. A General Collection of AVSI II Metaphors C-1

Appendix D. REFINE Code Listings for AVSI II D-1

Bibliography BIB-1

Vita VITA-1

List of Figures

Figure		Page
1.1.	Domain-Oriented Application Composition Environment	1-2
1.2.	Architect with AVSI	1-5
2.1.	Contemporary Icons	2-2
2.2.	Highway Icons	2-3
3.1.	OCU Subsystem Diagram	3-2
3.2.	AVSI Application and Subsystem Icons	3-6
3.3.	AVSI II Application and Subsystem Icons	3-6
3.4.	OCU Model Subsystem Diagram	3-7
3.5.	AVSI Icons for the Logic-Circuits Domain	3-10
3.6.	AVSI II Icons for the Logic-Circuits Domain	3-10
3.7.	AVSI Import/Export Icon	3-15
3.8.	AVSI II Import/Export Subsystem Icon	3-15
3.9.	2-Input And-Gate in AVSI II	3-18
3.10.	Event Queue	3-19
4.1.	Software Layers Beneath AVSI	4-2
4.2.	Switch Icons	4-7
4.3.	INTERVISTA's Link Pencil	4-16
5.1.	Exclusive-OR Circuit and Truth Table	5-4
5.2.	Half Subtracter Circuit and Truth Table	5-5
5.3.	Binary Array Multiplier Circuit and Truth Table	5-6
5.4.	BCD to Excess-3 Decoder Circuit and Truth Table	5-7
A.1.	BCD to Excess-3 Decoder Circuit	A-1

Figure		Page
A.2.	AVSI II Control Panel	A-2
A.3.	Edit-Application-Window	A-4
A.4.	Edit-Application-Window	A-5
A.5.	Edit-Update-Algorithm Windows	A-6
A.6.	Edit-Update-Algorithm	A-7
A.7.	Subsystem Window	A-8
A.8.	Edit-Subsystem and Technology-Base Windows	A-9
A.9.	Edit-Subsystem-Window	A-11
A.10.	DRIVER's Import-Export Window	A-13
A.11.	Repositioned Icons	A-14
A.12.	Import Area for BCD-EXCESS3	A-14
A.13.	DRIVER's Import-Export Window	A-16
A.14.	Import-Export MSP-Windows	A-17
A.15.	DRIVER's Update Algorithm	A-18
A.16.	BCD-EXCESS3's Import-Export Window	A-19
A.17.	BCD-EXCESS3's Import-Export Window	A-20
A.18.	BCD-EXCESS3's Import-Export Window	A-21
A.19.	BCD-EXCESS3's Update Algorithm	A-22
A.20.	BCD-XS3 Application	A-26
B.1.	The Icon Edit Utility	B-7

List of Tables

Table | Page
2.1. Best Color Combinations 2-12
2.2. Worst Color Combinations 2-12
5.1. Exclusive-OR Test Results 5-4
5.2. Half Subtracter Test Results 5-5
5.3. Binary Array Multiplier Test Results 5-7
5.4. BCD to Excess-3 Decoder Test Results 5-8
A.1. BCD to Excess-3 Decoder Truth Table A-23

AFIT/GCS/ENG/93D-04

Abstract

This research refined the visual presentation and usability of a previously developed visual interface for a domain-oriented application composition and generation system. The refined visual interface incorporated domain-specific bit-mapped graphics and sophisticated user interface design concepts to reduce user workload. User workload was reduced through object layout, window design, and color utilization techniques; by combining repetitive procedures into single commands; and reusing, rather than recreating, composition information throughout the application composition process. The Software Refinery environment, including its graphical interface tool INTERVISTA, was used to develop techniques for visualizing and manipulating objects contained in a formal object base. INTERVISTA was supplemented with graphical routines provided by Common Windows, a Lisp-based graphical environment that serves as the foundation for INTERVISTA. The interface was formally validated with a well-understood application domain, digital logic-circuits, and users of the interface were polled to ascertain the subjective usability of the interface. A comparative analysis of the application composition process with the previous visual interface was conducted to quantify the workload reduction realized by the new interface. Workload was measured as the number of user interactions (mouse or keyboard) required to compose an application. On average, application composition effort was reduced 44.0% for the test cases.

Developing a Sophisticated User Interface to Support Domain-Oriented Application Composition and Generation Systems

I. Introduction

1.1 Background

As quality computer hardware becomes more available and affordable, computer users are demanding more powerful, high quality, affordable software. The principles of software development are undergoing substantial evolution as software engineers and programmers endeavor to comply with the demand. Software engineers are embracing traditional engineering practices by introducing the concepts of reusable software components, object-oriented designs, and application domain-specific models. The destiny of software development is to evolve from a hand-crafted art into a true engineering discipline. As with other engineering disciplines, software engineers must develop understandable and verifiable components that can be combined into a cohesive whole, often by the users of a software system. Essentially, in the future software engineers will deliver the knowledge – domain-specific knowledge – about how to generate software, rather than delivering software itself (16:73). The software generation capability will be delivered in the form of program composition environments or application generators. A notional view of a domain-oriented application composition environment is shown in Figure 1.1.

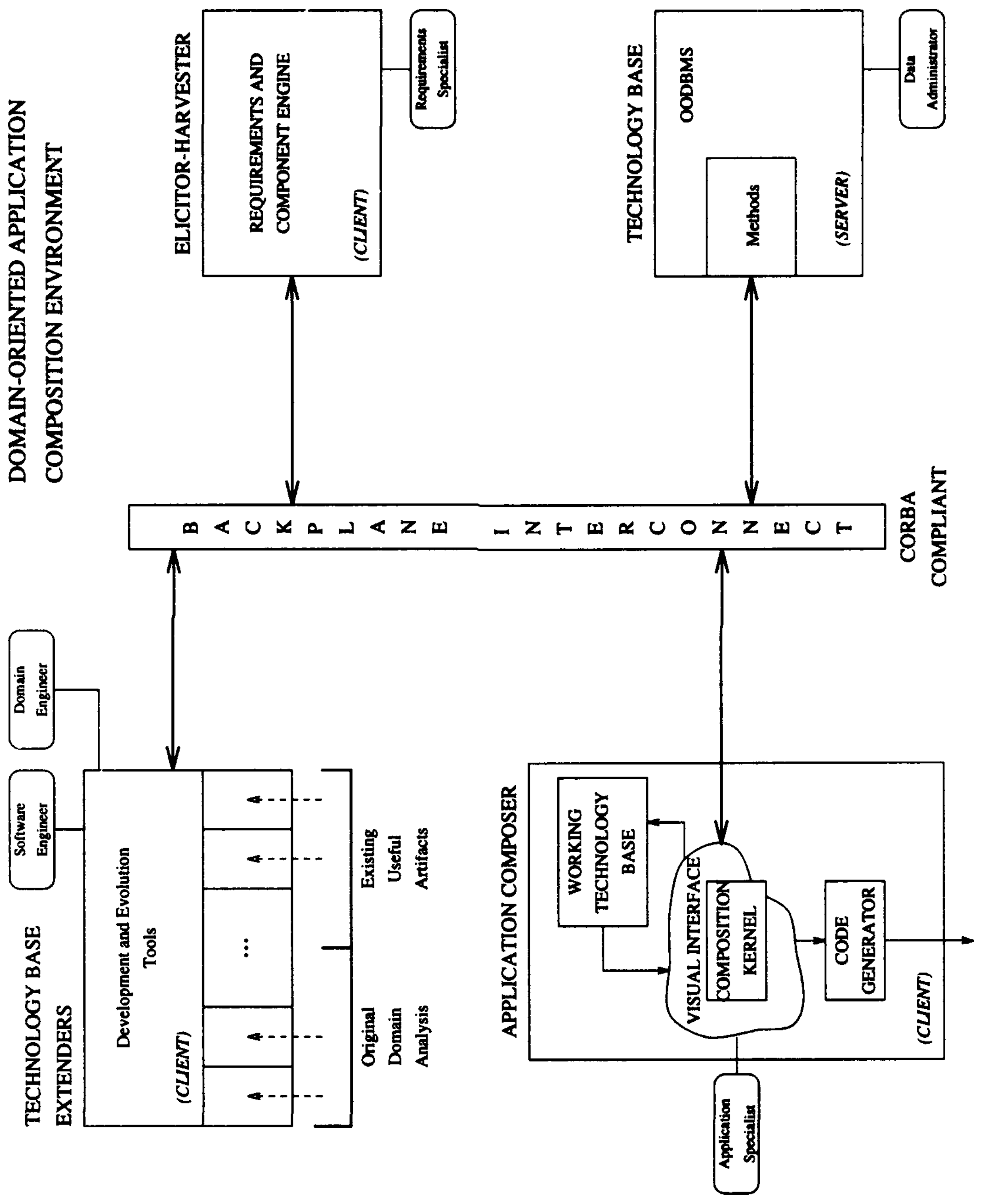


Figure 1.1 Domain-Oriented Application Composition Environment

The Knowledge-Based Software Engineering (KBSE) research group at the Air Force Institute of Technology (AFIT) continues to conduct research into the various aspects of domain-oriented application composition shown in Figure 1.1. A prototype domain-oriented application composition system was the subject of research by the KBSE group. The system, called Architect, was developed by Anderson and Randour (1, 22) and has subsequently undergone evolutionary development. Architect runs within the Software Refinery^{TM} development environment, centered around a Lisp-based specification language called REFINE. Architect provides an application specialist (the intended user) with the means to compose applications within specific domains. Conceptually, a user identifies a validated requirement to an analyst who develops an operational concept for a system to fulfill the requirement. The concept is then broken into software and hardware subsystems. Then, based on detailed knowledge of the target system and operational environment provided by a domain engineer, relevant software components are mapped into a domain model and a formal object or technology base by a software engineer. The user, or application specialist, manipulates the domain model to develop a preliminary software specification within Architect. The specification can then be compiled, and the resultant code can be "executed" to simulate system operation. The execution is monitored for desired functionality of the "specified system." The development/compilation/execution can be re-accomplished until the specification's behavior mirrors the system's desired behavior. A formal specification can then be generated and eventually translated into a deliverable software application in Ada, C, or another language.

The Architect system development process briefly described above can be quite involved and complex. A rudimentary visual interface was developed to replace Architect's original text-based command line interface, thus making system use significantly easier (33). The visual system, called Architect Visual System Interface (AVSI), allows the application specialist to "see" the application on the screen rather than having to visualize it in his head and to manipulate it as a graphical abstraction. Figure 1.2 shows the relationship between AVSI and the other components of Architect. AVSI is an object-oriented, direct manipulation (DM) interface. The term "object-oriented DM" refers to a class of interfaces that rely on "concrete and visible objects [icons] . . . where the key activity is visibly moving objects by pointing at them"(30:365). Although AVSI was a significant achievement, its usefulness was primarily functional. However, the manner in which a software system displays information to, and interfaces with, its user either invites errors or helps minimize them (8:95). Therefore, the aesthetic enhancement and usability, as well as continued functional development, of the Architect user interface was the goal of this thesis effort. Readers unfamiliar with the basic AVSI are referred to Weide's thesis, "Development of a Visual System Interface to Support a Domain-oriented Application Composition System" (33).

1.2 Problem

The reason for using a graphical interface is fairly straightforward. Simply put, it allows the application specialist to interactively participate in the application development process without having to become a programmer (35:78). A Graphical User Interface (GUI) has the benefit of allowing novices to learn a system's basic functionality quickly, usually through a demonstration from a more experienced user (24). It can display information in understandable and recognizable desktop "metaphors" or icons representing the actions,

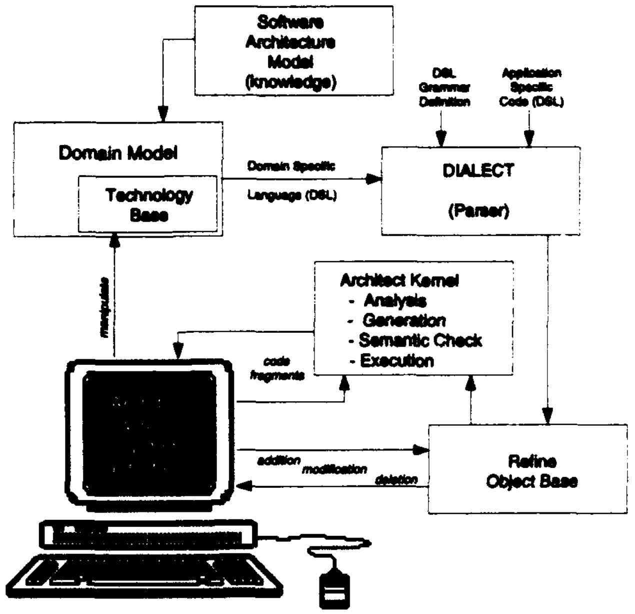


Figure 1.2 Architect with AVSI

procedures, or structure of the underlying code and objects (6:67). For example, a folder icon can represent a file directory, or an icon designed to look like an AND-gate can represent the code fragments that simulate an AND-gate object.

AVSI was written in INTERVISTA, the graphics package companion to REFINE. Because INTERVISTA provides the means to visually manipulate the REFINE object base, AVSI provided the application specialist significant functional power. However, the graphical ꞏophistication of INTERVISTA is limited. In order to make Architect a "user friendly" tool for tbe application specialist, AVSI needed aesthetic enhancements.

INTERVISTA allows a user to create and manipulate four basic objects: surfaces, windows, links, and icons. A surface is a two-dimensionally infinite plane, whose origin is at (0,0), that serves as a base for the other graphical constructs. A window is a rectangular opening that exposes a part of the surface. A window is also called a viewport. A link is a line (curved or straight, dashed or solid) that is the visible manifestation of a connection between two icons. An icon is a two-dimensional, geometric shape that represents an object in the REFINE object base. Legal shapes for an INTERVISTA icon are box, ellipse, diamond, and text. Size and height-width-ratio can be adjusted for an icon, giving some small variety in available icons. A text icon is simply an icon label without the icon.

These four visual objects, and a fairly robust set of functions to manipulate the objects, are the extent of INTERVISTA's graphical sophistication. With these graphical limitations in mind, developing AVSI II (an enhanced AVSI) offered a number of challenges. First, AVSI utilized the full potential of INTERVISTA's graphics. To be effective, AVSI II's graphical metaphors, used to represent objects in an application domain, had to be representative of the domain being *manipulated*. This is something that is beyond the INTERVISTA graphics capability. A means of augmenting INTERVISTA's graphical capacity was required. REFINE and INTERVISTA are implemented in Allegro Common Lisp, and several graphical interface development packages exist for Lisp-based GUI's. Additionally, INTERVISTA is built on top of Allegro Common Windows^{TM}, an interface from Lisp to the general window system resident on the hardware system. Common Windows provided an easy extension to INTERVISTA's graphics.

Selection of appropriate metaphors was a second challenge. Each domain model of interest requires a different set of "representative" icons. Similarly, a particular icon or metaphor might not be appropriate across different levels of abstraction within the same object domain. With careful design, it is possible to represent large amounts of information without creating a cluttered display. Iconic design and metaphoric selection, as well as spatial relationships and screen layout, was a significant portion of this research effort.

Lastly, although AVSI provided an application specialist with a broad base of functionality, some of the processes were repetitive, time consuming, or tedious. The challenge was to make AVSI II more usable without changing, or at the least degrading, the basic functionality of AVSI.

In order to meet the challenges described above, the following research goals and objectives were established:

1. Develop a set of domain-specific graphical icons (metaphors) for an existing Architect application domain and integrating the graphics into AVSI II's application construction process.
2. Develop the means for domain engineers to develop and incorporate additional iconic metaphor sets for new application domains.
3. Redesign the update algorithm definition process to eliminate duplication of effort and tedium.
4. Redesign the import/export connection process by exploiting the domain-specific graphics to make object connection more intuitive.
5. Support the visualization needs of concurrent research activities.
6. Test the new interface and quantify AVSI II's improvement in the areas of usability and composition effort reduction.

1.3 Assumptions

Several assumptions were made for the development of AVSI. Since the functionality of Architect and AVSI were validated using the digital logic-circuits domain, it was assumed that the domain was adequate for testing and validation of AVSI II. Additionally, the basic functionality of AVSI was considered adequate and no significantly new capabilities were developed. Finally, it was assumed that reuse of the existing AVSI would be maximized to maintain compatibility with other research efforts on Architect and with previously developed applications. As such, REFINE and INTERVISTA were the primary development packages for AVSI II.

1.4 Scope

AVSI II was intended to extend the *graphical sophistication of the Architect interface*. Therefore, no significantly new functional features were implemented. The scope of AVSI II was limited to enhancement of the visual presentation of information to a user and simplification of the application specialist's design effort. AVSI II does not provide:

- Visual support to the semantic check process or the application execution. These topics are, however, logical extensions in the development of more sophisticated interfaces.
- More than rudimentary support for an Application Executive. Full support of time-driven simulations or event-driven simulations was not incorporated but should be a topic for follow-on research.

1.5 Sequence of Presentation

The remainder of this thesis is organized as follows:

Chapter II provides a review of current literature concerning techniques for icon and metaphor design, window layout considerations, and uses of color in user interfaces. A survey of Lisp-based graphical interface software packages and techniques is also included.

Chapter III illustrates the operational concept for AVSI II by tracing through the application composition process. The shortfalls of AVSI are analyzed at each step, and the changes implemented in AVSI II are discussed.

Chapter IV discusses the specifics of each aspect of AVSI II's design, design tradeoffs, and implementation details.

Chapter V addresses the testing and validation of AVSI II and presents an analysis of its capabilities.

Chapter VI contains conclusions about the AVSI II research effort as well as recommendations for future research.

Several appendices are included to provide additional information concerning Architect and AVSI II. Appendix A provides a sample session with AVSI II wherein an application from the digital logic-circuits domain will be built. Appendix B presents detailed instructions for a domain engineer on designing and building a set of iconic metaphors compatible with AVSI II and incorporating them into the Architect system. Appendix C contains a description of a collection of general metaphors available for use with AVSI II. Appendix D consists of a list of the files required to run AVSI II and a point of contact from whom to acquire the REFINE source code for AVSI II.

II. Literature Review

2.1 Introduction

The objective of this literature review was to examine relevant issues and research in the areas of human-computer interaction and specifically address the issues of icon and metaphor design, window layout considerations, and design concerns in incorporating color into a user interface. The review involved an extensive gathering of data and an evaluation of that data in the context of applicability of the information and techniques to the AVSI II design effort. Although much of the information is general in nature, some references were found that addressed specific areas of the AVSI II design. Additionally, several commercial and public-domain Lisp-based GUI development environments and graphical support packages were reviewed as potential augmentation tools for INTERVISTA.

2.2 Icon and Metaphor Design

2.2.1 Icons in Everyday Use. Webster's New Collegiate Dictionary defines an icon as "a usually pictorial representation: IMAGE" (19). Although icons are frequently associated with religious imagery, an icon is, in fact, any picture or symbol used to represent another object, idea, or action. Icons have infiltrated almost every aspect of our daily lives, especially as a result of attempts to communicate across the barriers of language. A cross has been an iconic representation of the Christian religion for almost two thousand years; however, there are many examples of more contemporary icons. Even young children recognize a man's or woman's silhouette as representing a public restroom, a circle with a

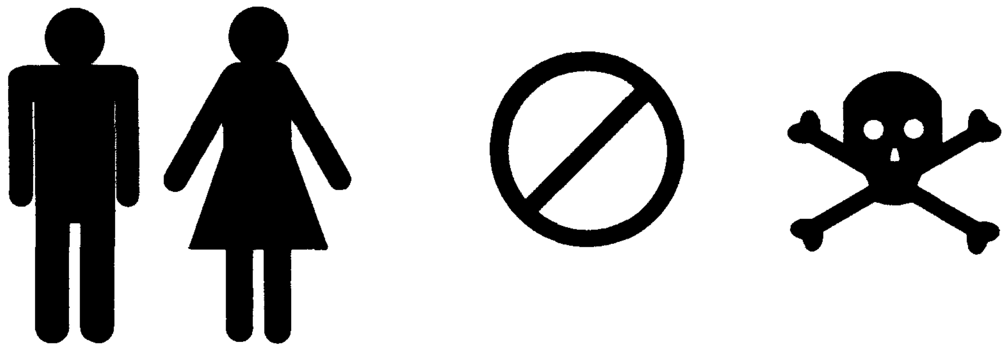


Figure 2.1 Contemporary Icons

bar through it as representing "don't," or a skull and cross-bones as representing poison (see Figure 2.1).

Perhaps the most widespread use of icons is in highway signs. Figure 2.2 shows some signs that convey precise messages simply through their shape or picture; any accompanying text is usually unnecessary. This capability to convey meaning is reflective of Arend's concept of a "global superiority" visual effect (2). The global superiority effect asserts that global features like shape, size, and color can be recognized and assimilated faster than features like lines and text. In other words, simple pictures with one or two distinctive features make effective icons. Nan Shu also endorses the use of icons by stating (25):

- Pictures are more powerful than words as a means of communication because they convey more meaning in a more concise unit of expression.
- Pictures aid comprehension and remembering.
- Pictures typically do not have language barriers. Properly designed pictures are understood by people regardless of their native language.

Given the extensive use of icons in our everyday lives, it is not surprising that the computer industry, always looking for ways to appeal to new users and cross the barriers

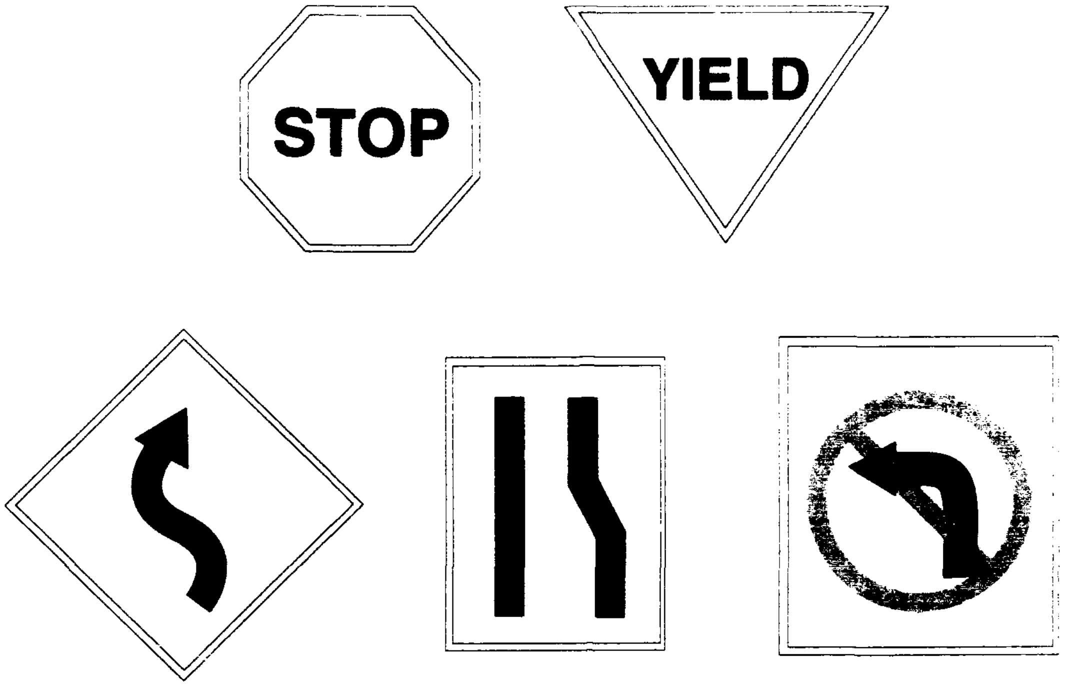


Figure 2.2 Highway Icons

of command/programming languages, is moving away from complex textual descriptions and commands and toward simpler, more easily understood iconic interfaces.

2.2.2 Icons in Computer Use. The primary goal of a GUI is to increase human productivity and speed, and to reduce human error and fatigue. In *Human-Computer Interface Design Guidelines*, Brown notes that appropriately applied icons and symbols, directly manipulatable by a user in ways analogous to reality, can help the user maintain context and orientation and reduce the user's need to memorize commands and syntax. More importantly, looking at and working with pictures is more fun than working with a screen full of text. Icons can be more effective than text because they can convey a large amount of information in little space, they are easily learned and easily remembered, and they are more universal (3). The icons and symbols on a display are frequently referred to as metaphors because of the designed intention of using them to represent a piece of reality in an abstract way. The icon or metaphor is a visual representation of a command, task, method, file, object, etc. The first challenge in GUI design is, therefore, to develop a set of metaphors appropriate to the domain of interest.

Determining what metaphor to use as an icon can be difficult. The key to developing a useful metaphor is to exploit a user's knowledge of familiar objects in a domain. Erickson recommends looking at a system's functionality for clues. He says to "note what metaphors are already implicit in the problem description [domain]" because people commonly use metaphors when talking about abstract concepts, and "it's almost certain that metaphors are lurking about in the description of the [system's] functionality" (9:69).

Many authors have attempted to describe what constitutes a good icon design. Deborah Mayhew recommends the following (18:316-330):

- Minimize semantic and articulatory distance,
- Provide visual feedback for icon position, selection, and movement,
- Choose a consistent icon scheme,
- Design icons to be concrete and familiar,
- Design icons in a set to be visually and conceptually distinct,
- Avoid excessive detail in icon design,
- Accompany icons with names,
- Limit the number of icon types to 12 if possible, but at most 20.

Two of these design features (i.e., provide visual feedback and name the icons) are incorporated into AVSI's design.

Semantic distance is the subjective or psychological distance between the user's intended action on an icon and the operations allowed by the interface to be performed on the icon. Semantic distance directly applies to the level of abstraction represented by the GUI. Articulatory distance refers to the relationship between an icon's shape and its meaning. For example, representing a Nand-gate with an icon that looks like a baseball imposes a large articulatory distance on the metaphor.

Ben Shneiderman recommends many of the same guidelines, but adds that the designer should (24:210):

- Make the icon stand out from its background,
- Ensure the harmoniousness of each icon as a member of a family of icons.

Mayhew's fifth guideline (i.e., design icons in a set to be distinct) and Shneiderman's second guideline (i.e., ensure the harmony of each icon in a family) relate well. Within a specific domain (or set or family) every icon must be distinct and recognizable from every other icon, but they must all relate to the overall domain. A paintbrush would be out of place in a logic-circuits domain, for example.

Carroll, Mack, and Kellogg state that developing a set of interface metaphors, or icons, is an iterative process that is refined over time. They claim the first step in creating a metaphor set is to generate a set of candidate metaphors and develop them using these ingredients:

- User's experiences with other systems or situations similar to the system being designed,
- User's knowledge, perceptions, and motivations,
- Sheer invention.

They further emphasize that they know of no formal analysis method for generating useful metaphors (6).

Chang goes a step further by recommending a strategy for creating visual representations of objects. He recommends considering the following factors (7:69-70):

- Application and Cultural Dependence: Icons typically represent a class of objects rather than one specific instance. Avoid excessive detail and reflect the semantic nature of an object.
- Easy Recognition: An icon should have a precise meaning that is easily recognized once identified. Icons do not have to be read, and people typically do not read words they are familiar with anyway (Note: This does not preclude icon labels). There are more distinctively shaped pictures than words.
- Distinction From Other Icons Within the System: An icon must be individually distinctive and should use a distinguishing feature to reflect the meaning it represents.
- Consistency in an Environment: Although icons should be distinctive, they should be consistent and harmonious within a system or domain.

Erickson describes a process for generating interface metaphors which consists of the following fundamental steps (9:68-72):

- Determine Functional Definition: It is necessary to understand something in order to develop a model/metaphor for it. This includes what it looks like, how it functions, and what its semantic behavior is.
- Identify User's Problems: Figure out what information about an object is important to a user and incorporate the information into the icon.
- Generate Metaphors: Note what metaphors are already implicit in the problem (or domain) description. Using previously identified metaphors can aid recognition of the icon.
- Evaluate Interface Metaphors: Once the metaphors have been developed, evaluate their accuracy and effectiveness through actual use. Allow users to provide feedback on the icon design.

2.3 Window Design and Layout

Thomas Tullis summarizes the importance of the overall window design and layout. He states that the overall content and format of the information in a display will have a substantial impact on the user's ability to interact with it. He asserts that if a system, or interface, does not present the information that a user needs in a format that is usable and understandable, the user's performance may be degraded or he may decide to abandon the system (29). Several other authors concur with the sentiment by explaining that increasing display density tends to decrease overall user performance. A well designed display (window or set of windows) should display all the information needed to perform a given task. It should also display only the information needed to perform a task; additional information may confuse a user and distract from the importance of the required information.

Shneiderman contends that a display format should be familiar to a user and should be reflective of the task being performed with the data. He echoes Tullis's concern for localizing relevant information to a single display by stating that users should not be required to remember information from one screen for use in another screen (24:79).

Additionally, Shneiderman advises caution in designing a direct manipulation (DM) interface. DM designs tend to consume valuable screen space and, as a result, valuable information is frequently available only by scrolling or opening another window. For experienced users of a system, a tabular display is often preferable to a graphical display (24:205).

2.4 Color Usage

The use of color in a computer interface can be separated into two distinct aspects. Color can be part of the information in the window (such as text or icons) or it can be part of the window itself (such as the background, frame, title, or control devices).

Color adds an entirely new dimension to a GUI. Color is easy to incorporate into a display and, subsequently, easy to misuse. The conservative use of color can provide powerful context and content cues to a user, while a busy and overly colorful screen may only confuse a user and mask the importance of information.

There is an abundance of information relating to color use in computer displays, and many guidelines have been developed. Shneiderman recommends that an interface designer (24:325-328):

- Use color conservatively,
- Limit the number of colors,
- Recognize the power of colors as a coding technique,
- Design for monochrome first, then add color judiciously,
- Be consistent in color coding,
- Be alert to problems with color pairings, and
- Use color in graphic displays for greater information density.

Mayhew endorses the same guidelines but adds her own recommendations (18:490-502):

- Be consistent with color associations people have in their jobs and cultures,
- Limit the number of colors to between four and eight,
- Choose symbol (icon) and background color combinations carefully, and
- Recognize that color use can improve user satisfaction.

Furthermore, Brown adds the following design considerations (5:67-76):

- Use color to highlight or emphasize,
- Do not use color to compensate for a poorly formatted display,
- Be aware of abnormal color vision,

- Use similar colors for related data, and
- Use contrasting colors to distinguish data.

Several items above refer to consistency and color associations. The importance of maintaining cultural color associations needs to be stressed. Different people use colors to mean different things, but there are several standard associations. For example, red means stop, hot, or danger; blue means cold, cool, or safe; yellow means warm or caution; and green means go. Violating these and other color associations could cause confusion for the application specialist.

One important color consideration mentioned above is color-blindness. Studies have shown that 8% of men and 0.5% of women are to some degree color-vision deficient, particularly in distinguishing green from red (23:271). This fact enforces the importance of careful color use.

Two themes recur in each author's design guidelines: design a display as monochrome first and choose appropriate color pairings. Designing a monochrome display first forces the GUI designer to carefully consider metaphor choice and icon shape and design, as discussed in Section 2.2.2. Color can then be conservatively and judiciously added to a display. Color should be used to enhance a GUI and highlight or emphasize similarities and differences. It is important to note that not all computer systems are equipped with a color display capability so the GUI must be usable without color.

Secondly, care must be taken with color pairing, or the choice of icon and background colors. Certain color combinations are compatible, while others are not. Poor color pairing contributes to user fatigue and difficulty in discriminating among icons. Designers should use high contrast color combinations to indicate differences and low contrast colors to indicate similarities. Following are some examples of effective color combinations (18:498):

High Contrast Colors	Low Contrast Colors
Red - Cyan	Yellow - White
Blue - Yellow	Red - Magenta
Green - Magenta	Yellow - Orange
Red - Green - Blue	Red - Orange
Red - Green - Blue - White	Blue - Cyan

2.4.1 Window Color. The effect of color on the usability of a display is a topic widely discussed in the literature: however, few hard facts are available. Murch presents data from a study in which 16 participants were asked to select the best and worst color combinations for text and lines against backgrounds. The data was collected for text and thin lines (two pixels wide) and for thick lines and panels. The results of the study are shown in Tables 2.1 and 2.2. Although the data-set is limited, it does address the importance of color choice for foreground and background colors in a display (21).

2.4.2 Icon Color. When and how to use color in icons are also considerations when designing an interface. Color can create strong associations when used in conjunction with certain shapes or locations (23:275). One effective use of color is to provide a user with information not otherwise available in the display or to redundantly reinforce information that is already there. However, the use of color in icon design is not discussed as extensively as is color use in the overall display. For icons, color is most effective as a highlight mechanism or as a method to convey groupings of data. Baecker effectively cautions

Table 2.1 Best Color Combinations (21)

Background	Thin Lines and Text		Thick Lines and Panels	
White	Blue (94%) Red (25%)	Black (63%)	Black (69%) Red (31%)	Blue (63%)
Black	White (75%)	Yellow (63%)	Yellow (69%) Green (25%)	White (50%)
Red	Yellow (75%) Black (44%)	White (56%)	Black (50%) White (44%)	Yellow (44%) Cyan (31%)
Green	Black (100%) Red (25%)	Blue (56%)	Black (69%) Blue (31%)	Red (63%)
Blue	White (81%) Cyan (25%)	Yellow (50%)	Yellow (38%) Black (31%) White (25%)	Magenta (31%) Cyan (31%)
Cyan	Blue (69%) Red (37%)	Black (56%)	Red (56%) Black (44%)	Blue (50%) Magenta (25%)
Magenta	Black (63%) Blue (44%)	White (56%)	Blue (50%) Yellow (25%)	Black (44%)
Yellow	Red (63%) Black (56%)	Blue (63%)	Red (75%) Black (50%)	Blue (63%)
	Overall Frequency of Selection			
	Black	25%	Black	23%
	White	20%	Blue	19%
	Blue	20%	Red	17%
	Yellow	13%	Yellow	13%
	Red	11%	White	10%
	Cyan	5%	Magenta	7%
	Magenta	4%	Cyan	6.4%
	Green	1%	Green	4%

Table 2.2 Worst Color Combinations (21)

Background	Thin Lines and Text		Thick Lines and Panels	
White	Yellow (100%)	Cyan (94%)	Yellow (94%)	Cyan (75%)
Black	Blue (87%) Magenta (25%)	Red (37%)	Blue (81%)	Magenta (31%)
Red	Magenta (81%) Green and	Blue (44%) Cyan (25%)	Magenta (69%) Green (37%)	Blue (50%) Cyan (25%)
Green	Cyan (81%) Yellow (37%)	Magenta (50%)	Magenta (69%) Yellow (44%)	Blue (50%)
Blue	Green (62%) Black (37%)	Red and	Green (44%) Black (31%)	Red and
Cyan	Green (81%) White (31%)	Yellow (75%)	Yellow (69%) White (56%)	Green (62%)
Magenta	Green (75%) Cyan (44%)	Red (56%)	Cyan (81%) Red (44%)	Green (69%)
Yellow	White and	Cyan (81%)	White (81%) Green (25%)	Cyan (56%)
	Overall Frequency of Selection			
	Cyan	24%	Cyan	23%
	Green	18%	Yellow	17%
	Yellow	16%	Green	16%
	Magenta	11%	Magenta	12%
	Red	10%	White	12%
	White	8%	Blue	9%
	Blue	8%	Red	7%
	Black	3%	Black	2%

against the overuse of color by admonishing the interface designer to avoid the "fruit salad" appearance that can be caused by the undisciplined use of color in a display (4:304).

2.5 Lisp-based Graphics Software

2.5.1 INTERVISTA. INTERVISTA^{TM} is a Lisp-based tool that allows a user to create interactive user interfaces for applications written in REFINE. INTERVISTA provides tools for constructing diagrams, pop-up menus, and mouse-sensitive text windows. Diagrams are created in an INTERVISTA window using icons (boxes, diamonds, ellipses) and links (connections). INTERVISTA provides the user with a capability to customize diagrams by altering icon shape, size, and labels; controlling link arrow-heads, labels, and type; and adjusting window size and shape. Pop-up menus can be single-item or multiple-item selections. Text windows are an extension of the REFINE printer and can contain regular or mouse-sensitive text. INTERVISTA is a commercial product sold as part of the Software Refinery^{TM} specification language environment by Reasoning Systems Inc. INTERVISTA is built on top of Common Windows, the underlying window system (13). Both AVSI and AVSI II are written in INTERVISTA.

2.5.2 Common Windows. Allegro Common Windows^{TM} provides the interface from Lisp to the general window system supplied with the hardware. Common Windows (CW) uses the X window system and acts as a front end to Common Lisp X interface (CLX). CW provides the capabilities to create and manipulate window streams, active-regions, regions, bitmaps, bitmap-streams, text operations, menus, and more. Multiple window environments and scrolling are supported. Graphics options include bitmap and color manipulation. Common Windows is a commercial product available from Franz, Inc. (10)

2.5.3 CLX. CLX (Common Lisp X interface) provides basic Common Lisp/X functionality. Each programming language must have a specific functional interface for using the X protocol. CLX is a de facto standard low-level interface to X-Windows for Common Lisp. CLX is a set of data types, functions, and macros that allow a program to interact with an X11 server. (27)

2.5.4 CLIM. Common Lisp Interface Manager (CLIM) is a high-level, portable user interface management system toolkit. It provides facilities for basic windowing, input, output, and graphics services. Additionally, it provides an Applications Programmer Interface (API) to user interface facilities for the Lisp application programmer. It uses the services of the host machine's window system (i.e., Motif or OpenLook) to integrate Lisp applications into the host's window environment. CLIM has the flexibility to provide detailed, low-level user control of interface layout as well as high-level automated layout capability. CLIM is commercial software developed by Symbolics. (26)

2.5.5 GINA. The following description is extracted from the Generic INteractive Application (GINA) user's manual (20):

> GINA is an object-oriented application framework written in CommonLisp and CLOS [Common Lisp Object System]. It is based on CLM, an interface between CommonLisp and the OSF/Motif software. The Generic INteractive Application is executable and has a complete graphical user interface, but lacks any application-specific behavior. Version 2.1 of GINA is publicly available and includes full documentation. It runs on Allegro or Lucid Common Lisp and on Symbolics Lisp Machines. A version of GINA for C++ is also available. . .

> CLM is a language binding for Common Lisp and OSF/Motif. It consists of a Motif daemon and a Motif server. The Motif daemon runs on an arbitrary machine in the network and listens for Lisp processes requesting to use CLM. The Motif daemon forks Motif server processes which communicate with their Lisp processes over a TCP/IP connection. When supported by the Lisp system, the Motif server can also be directly forked from the Lisp process. The Motif server offers the functionality of the X toolkit and the convenience functions of OSF/Motif on a network-transparent TCP/IP protocol. A package of Common Lisp functions provides a high-level interface to this protocol.

GINA and CLM are available free via anonymous ftp from ftp.gmd.de.

2.5.6 CLUE. The Common Lisp User Interface Environment (CLUE) was developed by Texas Instruments Incorporated. The following is extracted from TI's CLUE documentation (28):

> The Common Lisp User Interface Environment (CLUE) is a portable system for user interface programming in Common Lisp. CLUE, which is based on the X Window System and the Common Lisp Object System (CLOS), extends the CLX [Common Lisp X] interface to provide an architectural model for the construction of interactive Lisp applications. Modelled on the Xt toolkit library, CLUE could be described as a translation of the Xt "intrinsics" into the domain of Common Lisp and CLOS.

CLUE is available free by anonymous ftp from csc.ti.com.

2.5.7 Garnet. Garnet is a GUI with many high-level design and development features. It does not depend on CLOS, but does depend on CLX. Garnet (version 2.0 and after) is public domain and has no licensing restrictions. It runs in virtually any Common Lisp environment, including Allegro, Lucid, CMU, and Harlequin Common Lisps. (14)

2.5.8 LispView. LispView is a GUI written at Sun that does not use CLX. It converts Xlib.h directly into Lucid Lisp function calls. LispView is a generic Application Programmer's Interface (API) that provides Lisp programmers with a powerful set of tools to build sophisticated GUIs. It supports output functions for windows as well as a sophisticated input model for handling events. LispView provides sets of generic functions for reading and writing the state of an object. LispView can also translate an interface developed with the OpenWindows^{TM} Developers' Guide (Devguide) Interface Builder. It is available for anonymous ftp from export.lcs.mit.edu. (14)

2.5.9 Winterp. WINTERP (Widget INTERPreter) was developed at Hewlett-Packard and uses the Xtoolkit and Motif widget set. It is a free-standing Lisp-based tool for setting up window applications. WINTERP enables rapid prototyping of GUIs through the interactive manipulation of user interface objects and their attached actions. WINTERP supports rapid prototyping of GUI-based applications by allowing the user to interactively change both the UI appearance and application functionality. WINTERP is available via anonymous ftp from export.lcs.mit.edu. (12)

2.5.10 XIT. X User Interface Toolkit (XIT) is an object-oriented user interface development environment for the X Window System based on Common Lisp, CLOS, CLX, and CLUE. It was developed at the University of Stuttgart. XIT contains user interface toolkits, including general building blocks and mechanisms for building arbitrary user interface elements. It also has a set of predefined common elements (widgets), as well as high-level interactive tools for constructing, inspecting, and modifying user interfaces by means of direct manipulation. XIT is currently under active development, but is available by anonymous ftp from ftp.informatik.uni-stuttgart.de. (14)

2.6 Conclusion

Although much has been written about designing human-computer interfaces and many guidelines have been developed, a cookbook recipe for building a GUI does not exist. The most critical GUI design steps are the choice of metaphors appropriate to the domain of interest and the selection of icons to convey the metaphors. Well-designed icons make an interface intuitive and pleasant to use. Judicious addition of color to an interface can enable the application specialist to easily key on critical relationships and to maintain an awareness of his situation. The design considerations presented in this chapter were taken to heart and are evident in several aspects of the design of AVSI II.

III. System Enhancements

3.1 Introduction

AVSI was completely adequate as a proof-of-concept visual programming tool. However, as Architect has evolved into a more sophisticated application composition and generation system, the visual aesthetics of the interface directly impact the system's usability. As stated in Chapter II, if a graphical interface is not easier to use than the alternative (say, a command line interface), the graphical interface will not be used. Therefore, improvement of the presentation and usability of AVSI was the goal of this research effort.

The structure of an Architect application is based on the Object Connection Update (OCU) Model developed by the Software Engineering Institute (SEI) (15). The OCU model consists of several software elements used to express a design in the form of subsystems and hardware interfaces. A subsystem, visualized in Figure 3.1, consists of a controller, a set of objects, an import area, and an export area. A software system, or application, is described as a set of subsystems controlled by an executive function. An executive, then, is a high level supervisory subsystem that coordinates the behavior and information flow between subordinate subsystems. The OCU model treats subsystems as self-contained, abstract units; however, there are no restrictions against a subsystem acting as an object controlled by another subsystem. The components of a subsystem, mentioned above, are:

- Controller: The controller aggregates a set of objects and manages the connections between them and the flow of information to and from them, based on the function or purpose of a subsystem.

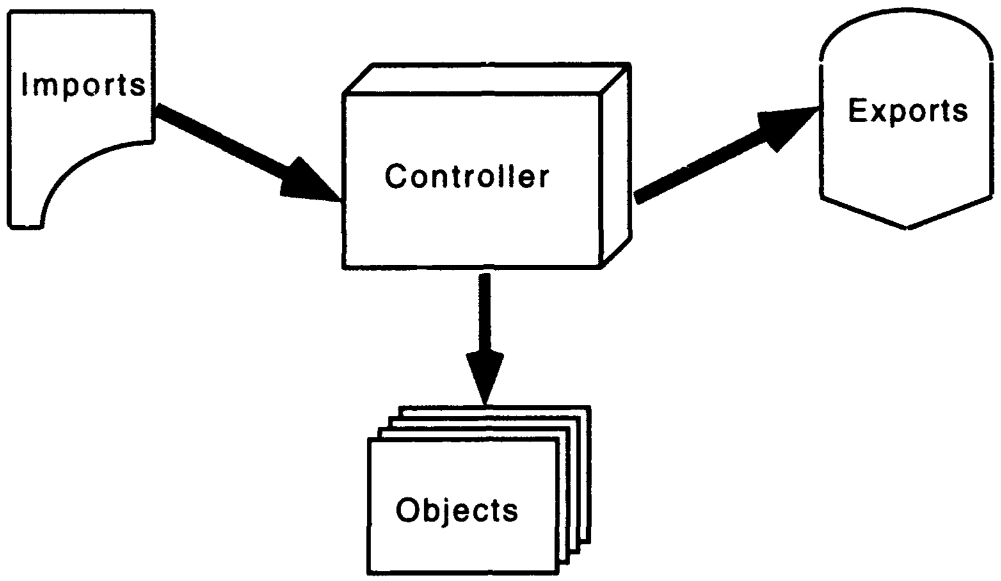


Figure 3.1 OCU Subsystem Diagram

- Import area: The import area is the central collection point for the inputs needed by the objects in a subsystem. The import area collects data from other subsystems in an application and makes it available to the objects.
- Export area: The export area is the central distribution point for the outputs of a subsystem that is needed by other subsystems in an application.
- Object: An object models the behavior of real-world components. An object maintains a state and has an associated set of algorithms responsible for transforming input data into the state data.

The visualization philosophy used in the development of AVSI II was to emulate and graphically display the application composition process in a manner consistent with the OCU concepts, as well as incorporating the domain-specific graphical representations.

3.2 Operational Concept

The application specialist must proceed through several steps to develop a complete application and simulate its execution or operation through AVSI. First, he selects the application domain, creates the application, and defines a controlling subsystem. Second, the application specialist incorporates lower level subsystem and primitive objects into the application. Next, he defines the order in which each primitive object in the application is updated (using the object's particular update algorithm). Lastly, the application specialist defines how the primitive objects communicate by connecting each object's export (or produced signal) to the appropriate object's import (or consumed signal). Once these steps are taken, the application specialist has fully defined the application's composition and behavior. Execution of the application can then be simulated to compare its behavior to the behavior expected from the design.

AVSI already provides a means to perform each step outlined above; however, the actual process and appearance of the interface was improved in several significant ways. The following subsections describe each of the earlier mentioned steps in more detail, covering the limitations of AVSI's original design and outlining the improvements incorporated into AVSI II. The presentation of information is roughly in the order it was accomplished. The enhancement to AVSI was accomplished as a stepwise refinement, with one area of enhancement being completed before expanding into another area. This process, although somewhat given to duplicative effort, ensured a fully functional interface at every step of the development. The steps taken to design and implement the improvements are discussed in more detail in Chapter IV. For more information on the original design and basic functionality of AVSI, see Weide (33).

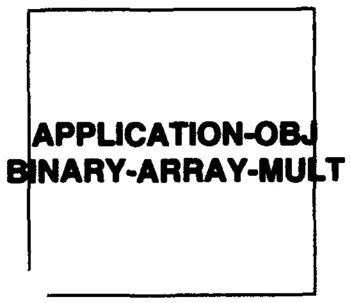
AVSI II uses a control panel window as the central interface to Architect. The control panel consists of an array of buttons and a message window. The buttons are used to create and edit applications, edit subsystems, define an application's import and export area connections, and initiate execution of the application.

3.2.1 Defining the Application. The application specialist creates an application-object in this step by selecting "Create New Application" from the control panel. He is then prompted for the application domain, the application name, and the execution mode of the application. The execution mode selection is a new feature of AVSI II. It was required to support the Application Executive developed by Welgan (34). The application-object has no functional capability and serves as little more than an identifier for the application being composed. Therefore, once the application has been created, the application specialist selects "Edit Application" from the control panel in order to create a controlling (or supervisory) subsystem for the application. Selecting the button exposes the edit-application-window, which contains a single icon representing the newly created application. The subsystem is created by clicking the mouse button on the surface of the edit-application-window and selecting "Create New Subsystem" for the pop-up menu. Once the subsystem is created and named, it is linked to the application by clicking the mouse on the subsystem icon and selecting "Link to Source" from the pop-up menu. The link is then created. The process is efficient and straight-forward. Subsequently, there was very little room for improvement in the basic interface method.

Due to INTERVISTA's limited choice of icons, the application object and subsystem locked the same in the edit-application-objects window. The only difference in the objects was the icon label (Figure 3.2). Bit-mapped images with more descriptive size, shape, and design were developed to supplement the INTERVISTA icons. The new bit-mapped icons incorporate the principle of global superiority introduced in Section 2.2.1. Figure 3.3 depicts the full-size and half-size icons developed for the application and subsystem objects. The distinctive size and shape of the icons make them immediately recognizable without the need to read the label.

In AVSI, no color (other than black and white) was used. An additional enhancement to the edit-application-window was to change the background color of the window to light blue. Section 2.4.1 outlined the power of using color as a visual cue to the user of a system. In this case, a glance at the screen to see the light blue is enough to tell an application specialist he is in a window where application or subsystem building is done. The color blue was selected because it is often associated with the sensations of cool or peaceful. The intent of using blue was to reduce anxiety or relax the application specialist during the creation of the application.

3.2.2 Building the Subsystems. Once the application has been created, the application specialist needs to create the lower level subsystem-objects and primitive-objects that constitute the application. Upon selecting "Edit Subsystem" from the control panel, the application specialist is presented with a window that represents the subsystem-object as a subsystem diagram from the Software Engineering Institute (SEI) Object-Connection-Update (OCU) model (15:18). AVSI displayed the diagram using two diamond shaped



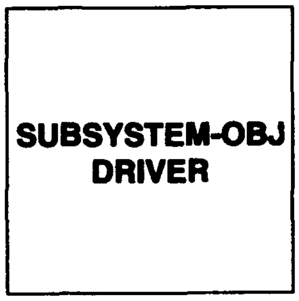


Figure 3.2 AVSI Application and Subsystem Icons

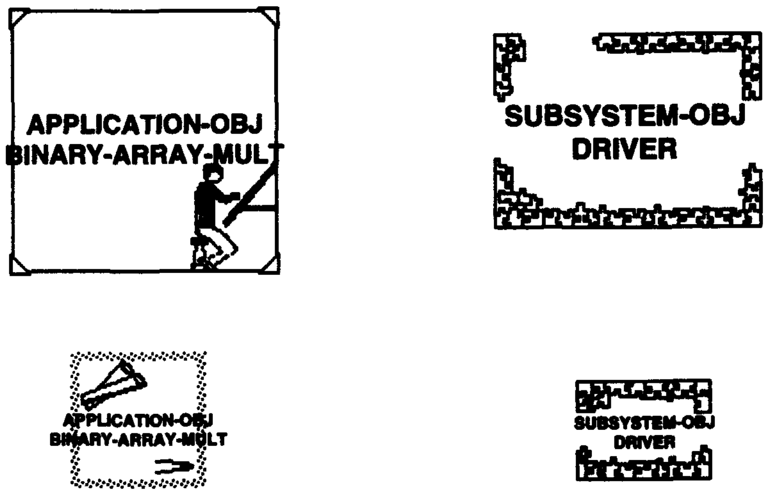


Figure 3.3 AVSI II Application and Subsystem Icons

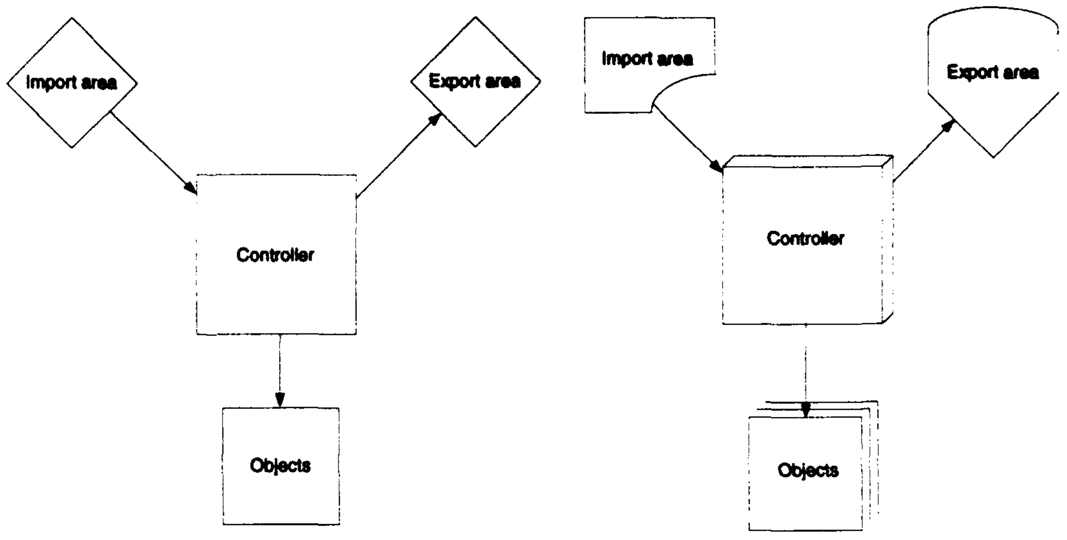


Figure 3.4 OCU Model Subsystem Diagram

icons and two box shaped icons. Using bit-mapped icons, AVSI II displays the diagram using the same shapes as the SEI. Although there is no functional difference between the representations, AVSI II's display is more readily recognizable as an OCU model. Figure 3.4 shows both representations of the subsystem diagram. This window was not given a colored background. The intent of color in AVSI II was to draw attention to significant windows and action areas. The subsystem window is an intermediate window where no design activity occurs.

By clicking a mouse button on the "Objects" icon, the application specialist opens two windows where he creates new subsystem-objects and primitive-objects for the application. The leftmost window, called the edit-subsystem-window, initially contains a single subsystem-object icon. The rightmost window, called the technology-base-window, contains a set of icons representing the primitive objects that are valid within the current application domain. The application specialist can add new subsystems to the application by clicking a mouse button on the edit-subsystem-window background and selecting the appropriate option from the resulting menu, or add primitive-objects by clicking a mouse button on the object in the technology-base-window and clicking the mouse again on the location in the edit-subsystem-window where the primitive should be placed. As each icon is placed in the edit-subsystem-window, it is linked to an appropriate subsystem by clicking the mouse on the icon, selecting "Link to Source" from the menu, and clicking the mouse on the source subsystem.

A functional enhancement was made in AVSI II that allows the application specialist to make multiple links at one time. By clicking the mouse on a subsystem, the application specialist can select "Link Multiple Targets" from a menu and be presented with a list of all icons in the window that are not currently linked to a subsystem. He can then select all, some, or none of the objects and the interface will create all indicated links at once. As an example, to connect six primitives to a subsystem under AVSI, the application specialist must perform 18 distinct mouse actions. Using AVSI II, the application specialist can perform the same task with only four mouse actions. The multiple link capability greatly enhances the interface's usability.

The visual enhancements to this portion of the process also involved icon improvement and the use of color. The improved icon used for subsystem-objects was discussed in the previous section. The primitive-objects in Architect and AVSI II are, for the most part, unique to a particular domain. Therefore, the icons used to represent the primitives should be unique as well. Additionally, the icons should have some associated meaning. The metaphor design concepts discussed in Section 2.2.2 have direct bearing on the icons developed for use with AVSI II. Whereas AVSI was restricted to box, diamond, and ellipse shapes to iconically represent the primitive-objects, AVSI II takes advantage of metaphors inherent in an application domain to "show" the application specialist which primitives are available. Figures 3.5 and 3.6 illustrate the icons used in AVSI and AVSI II, respectively, for the primitives in the logic-circuits domain of Architect. It is easy to understand how the AVSI II icon set shortens the application specialist's task of locating and selecting a specific primitive-object from the technology-base-window. Chapter IV addresses the specific design considerations that went into developing the logic-circuits set. Appendix B provides a detailed process for creating AVSI II icons for an application domain's primitive metaphors.

Both the AVSI and AVSI II icon sets present an icon as a shape with the primitive object-class name as a label. In AVSI the label is necessary for the application specialist to know which primitive a particular icon represents. AVSI II retains the convention of providing the object-class name as the label. Although a simple domain like the logic-circuits domain (used throughout this document as an example) makes the label seem like redundant information, a more complex domain may use more abstract metaphors as icons. The label serves to associate an icon with a primitive object. However, when a primitive is selected from the technology-base-window and added to the edit-subsystem-window, the application specialist assigns each instantiation of the primitive a name unique to the application. For AVSI, the primitive name is added to the object-class name. Without retaining the object-class name, confusion could result from the fact that, for example, the icons for a mux-, nand-, and nor-object look virtually identical. The bit-mapped images used as icons in AVSI II allow enough specificity and variability in the icons that the

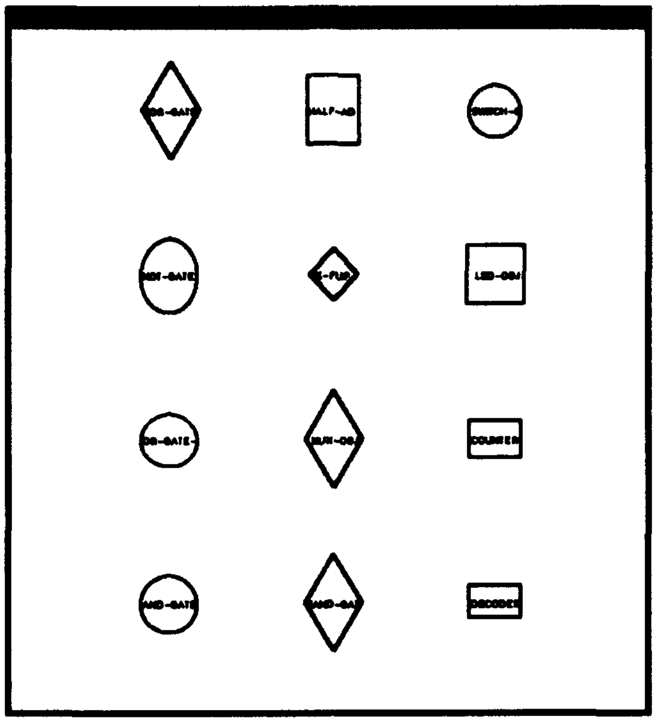


Figure 3.5 AVSI Icons for the Logic-Circuits Domain

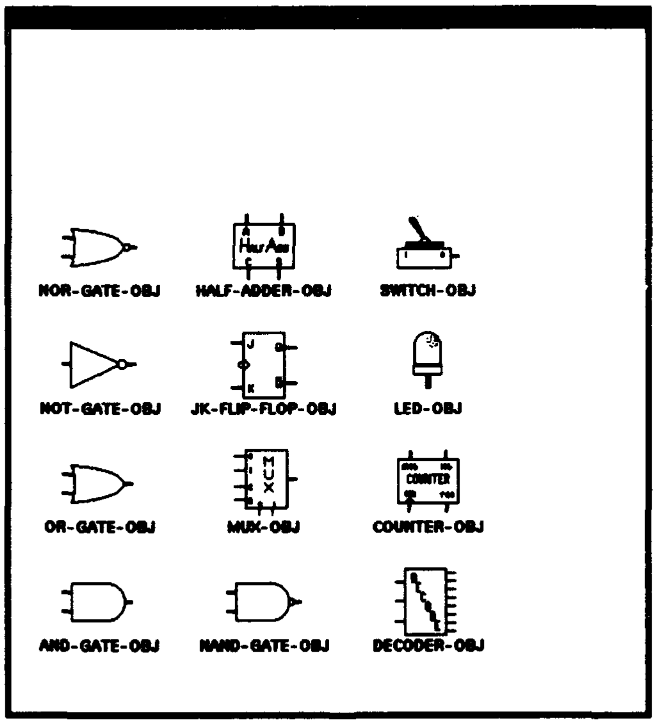


Figure 3.6 AVSI II Icons for the Logic-Circuits Domain

object-class name is no longer required and the name of the particular primitive instance is sufficient identification.

Color was used to provide the application specialist with basic situational awareness, in the same manner as described in the previous section. The edit-subsystem-window has the same light blue background as the edit-application-window to indicate that the windows have the same basic functionality. The technology-base-window was given a green background. The technology-base-window is the only green window in AVSI II, thereby making it immediately recognizable. Green is also a color culturally associated with safe or go. This is appropriate for the technology-base-window since there is no design activity in the window, it is simply a source of the primitives.

Color was not used in the design of the icons themselves. The primary reason was a desire to avoid the "fruit salad" effect mentioned in Section 2.4.2. Another reason was a fall-out of designing for monochrome first and adding color to a display later (Section 2.4). Once the icons were designed and incorporated into a monochrome AVSI II, there was no compelling reason to rebuild the icons in color. Additionally, Table 2.1 indicated that black was the preferred color for text and lines on colored backgrounds.

3.2.3 Specifying the Update Algorithm. An update algorithm is built for each abstract level of an application. For any subsystem, the update algorithm is defined for those objects (subsystem and primitive) that are controlled by, or linked to, the subsystem. At the application level, the process for creating an update is entered through a menu selection off the AVSI control panel. At a subsystem level, it is entered by clicking the mouse on the "Controller" icon in the current subsystem diagram window. With AVSI, the application specialist was presented two edit-update-algorithm windows: one giving a graphical (iconic) view of the update algorithm, and one giving the textual view. To add a primitive's update function to the algorithm, the application specialist had to click on a "nub" in the graphical window, select the type of update to be added (typically a call to a primitive's update function), and enter the name of the operand of the update call. Each operand is generically represented by a simple box icon and the operand name. It is important to note that the name of the operand in the update call must *exactly* match the instantiated name of the object in the application. Unfortunately, the edit-*-window that shows the names of the subsystem and primitive objects that apply to the update algorithm is not normally visible during this process (Note: edit-*-window is used in this chapter to indicate that the statement is equally applicable for the edit-application-window and the edit-subsystem-window). For a long update sequence, the update becomes tedious and presents numerous opportunities for the application specialist to incorrectly enter an operand name.

The procedure for building the update algorithm under AVSI II needed to be less tedious and less susceptible to mistakes. The key to simplifying the process was creating a new window, called the controllee-window, that displays all subsystem- and primitive-objects controlled by, or linked to, the current subsystem. The controllee-window displays a copy of the icons created in the edit-*-window and two additional icons that represent if-then and do-while constructs. To build the update algorithm, the application specialist clicks the mouse on an icon in the controllee-window. The mouse cursor changes to a "bullseye" shape, and the application specialist can click on a "nub" in the edit-update-algorithm window. AVSI II automatically adds the icon to the graphical update window, and the represented object to the textual update window. To create a six object update algorithm, AVSI required an application specialist to perform 42 distinct actions as well as remember the precise name of all six objects. Using AVSI II, the application specialist only performs two mouse actions per object, or a total of 12 actions. There is almost no source of error since the application specialist does not need to enter any operands or remember any names, thus reducing the time required to build an update.

One additional time saving feature was implemented for this process. During the construction of the update algorithm, three different windows are continuously exposed. The "Deactivate" menu choice was changed for the graphical update window to deactivate all three windows with a single selection, thus reducing the application specialist's workload from six actions to two.

As with the other AVSI II windows where graphical information is being displayed, the controllee-window and the graphical edit-update-algorithm window were given a distinctively colored background. In this instance, yellow was chosen based on the guidelines discussed in Section 2.4.1. Yellow, a color traditionally associated with caution in American society, was chosen for the update windows to convey that the actions taken to build an update algorithm are less flexible, and the order of the objects in the update is more critical, than the actions and order involved in linking primitives and subsystems in the edit-*-window. Color was not added to the textual update window, or any other text based window in AVSI II for the same reasons discussed in Section 3.2.2. The text based windows are primarily information only and no actions are required in them.

3.2.4 Connecting the Imports and Exports. The import objects (targets) and export objects (sources) are connected at two levels. Connections can be made within a subsystem or between subsystems. Using AVSI, the application specialist made the import-export connections by first clicking on the "Build Import and Export Areas" button on the control panel. A new window, the import-export-window, opened and displayed one or more three icon groups that represented all subsystems in the application and the subsystems' import and export areas (see Figure 3.7). The same icon group was used for representing primitive objects. The OCU model permits subsystems to be made objects (or subordinate subsystems) of another subsystem. With AVSI, if a subsystem controlled other subsystems, the lower level subsystems' connections were made by clicking on the subsystem icon's central box and selecting "Make Internal Connections" from the resulting menu. This action opened an import-export-window containing the lower level subsystem's controllees.

To make the connections, the application specialist selected an import or export area by clicking on it. A text window listing the import or export objects in that particular import or export area appeared. The application specialist then selected an import or export object with the mouse, and a set of dashed links appeared that indicated the areas that contained possible sources or targets with which to connect. He then clicked on a different export or import area icon (one indicated by a link), and selected an export or import object from the text window that opened. This process was repeated until all imports had been connected to at least one import and all imports were connected to one export.

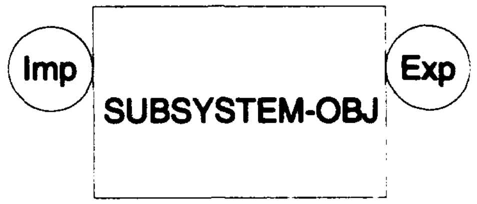


Figure 3.7 AVSI Import/Export Icon

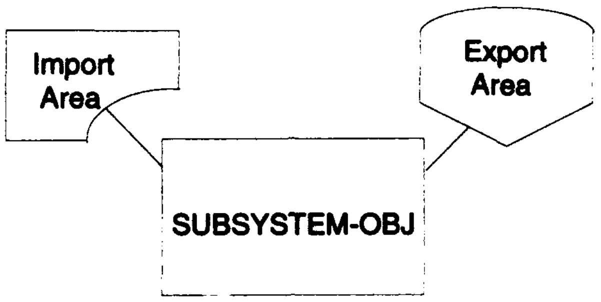


Figure 3.8 AVSI II Import/Export Subsystem Icon

Because of AVSI's abstract representation of the import and export areas, the application specialist had no direct feedback about which specific import and export objects were connected. The application specialist only knew that something in one area was connected to something in another. The abstraction makes sense for subsystem objects that are abstract by nature, but it is a hindrance to understanding the connections between primitives. Additionally, the primitive-objects were not presented to the application specialist. Rather, listings of the primitive's imports and exports were displayed. The import-export-window provided no visual feedback about which icon groups represented a specific primitive and which represented subsystems. Improvement to the visual presentation in the import-export-window included redesign of the subsystem icon group to look more like the OCU subsystem diagram discussed in Section 3.2.2 (see Figure 3.8). Further improvement was realized by displaying the specific primitive's metaphor in place of the icon group.

One visual representation of connections between subsystem import and export areas (like in the OCU model) is an abstract link between data areas. AVSI accurately portrayed the abstraction, however, primitive-objects, like the OCU model's objects, represent real-world entities (15:19). AVSI II re-implements the subsystem's internal connection process (i.e., that of connecting the primitive's imports and exports) by displaying the primitive's bit-mapped icon and allowing the application specialist to link specific import objects to specific export objects. AVSI did not display sub-subsystems and primitives of a subsystem in the same connection window. Instead, all subsystems in an application were displayed in one import-export-window, and the primitives controlled by a subsystem were connected by selecting "make internal connections." This brought up another import-export-window containing only the subsystem's primitives. AVSI II displays all objects controlled by a subsystem in the same window.

To make import-export connections in AVSI II, the application specialist begins by moving the icons from their default display locations to some "significant" position in the window. The default display locations are determined through a simple automatic layout algorithm. AVSI II does not have sufficient "knowledge" of the application's design to accurately place the icons; therefore, the application specialist has the option to move the icons. Section 2.3 discussed how the relative location of objects in a display can impact a user's ability to understand and use an interface.

Each primitive's icon consists of the bit-mapped image and label that identifies the primitive object, and a set of mouse sensitive 'fields' that correspond to the imports and exports defined for the primitive. For example, a 2-input And-gate primitive would have two import fields and one export field. Figure 3.9 shows the And-gate primitive with the import and export fields illustrated as grey outlines. Imports are connected to exports by selecting an export field with the mouse and then selecting the destination import field. The link is automatic and visible. The converse (i.e., connecting an export to an import) also holds. Unlinked import and export fields are displayed in reverse video to indicate to the application specialist that connections still need to be made. Once the connection is made, the connected fields disappear (un-highlight). The fields can easily be found from the termination of the links representing the connection. The overall effect of AVSI II's internal subsystem connection process is one of "hooking up" the primitives. Because the application specialist is typically not a programmer, it is important to reduce the abstraction of the import-export display to make the connection process more natural and intuitive.

Connecting a primitive-object to an object within another subsystem is a blend of the subsystem and primitive connection processes. Clicking on an subsystem's import or export area icon exposes a text window displaying the imports or exports of the subsystem. Clicking on an icon's field and then on a text window entry creates an import-export link. Alternately, selecting a text window entry and then clicking on an icon's field also creates the import-export link. The visual feedback is in the form of an updated text entry showing the connection and the field changing to an empty box that means "there is a connection made, but not to an object visible in the same window as the icon."

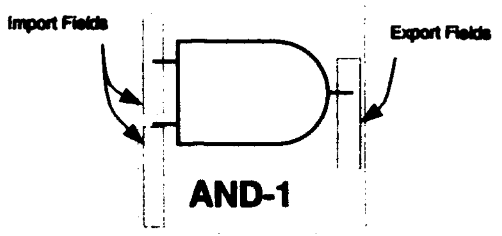
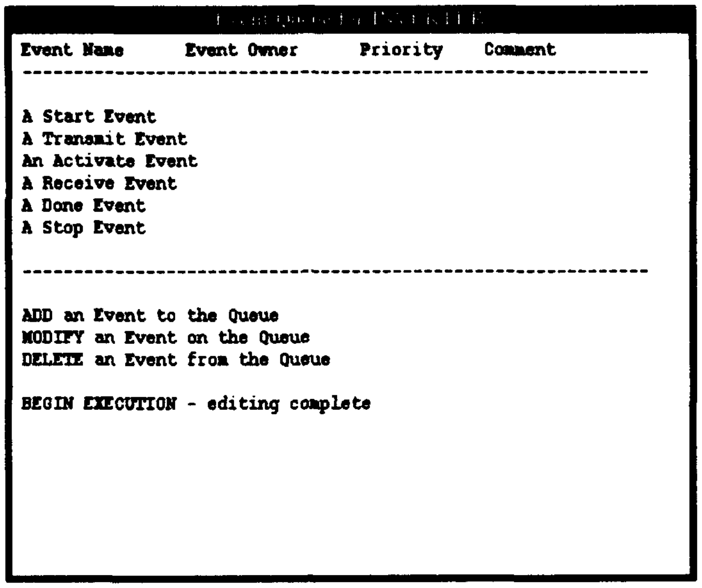


Figure 3.9 2-Input And-Gate in AVSI II

The ability to close multiple windows simultaneously was added to the import-export-window menu. The application specialist has a choice of closing only the import-export-window by selecting "Deactivate" from the window's title-bar menu, or closing the import-export-window and all exposed text windows simultaneously by selecting "Deactivate" from the window's surface menu.

Red was chosen for the background color of the import-export-windows. Culturally, red represents danger, heat, or warning. For AVSI II, red is intended to convey to the application specialist that the import-export connection process is the most complex and error fraught part of composing an application. Red is meant to instill a sense of caution and deliberation in the application specialist during the connection activity.

3.2.5 Executing the Application. Once the application has been built and semantically checked, the application specialist will likely execute it. This is accomplished by clicking on the "Execute Application" button on the control panel. If the application has a sequential execution mode, the execution simply proceeds and the results are displayed in the EMACS window. If the application has an event-driven execution mode, a text window displaying the current contents of the event queue is exposed. The window, shown in Figure 3.10, also contains four control selections that allow the application specialist



```
[illegible]
Event Name      Event Owner      Priority    Comment
------------------------------------------------------------

A Start Event
A Transmit Event
An Activate Event
A Receive Event
A Done Event
A Stop Event

------------------------------------------------------------

ADD an Event to the Queue
MODIFY an Event on the Queue
DELETE an Event from the Queue

BEGIN EXECUTION - editing complete
```

Figure 3.10 Event Queue

to add events to the queue through a set of input menus, modify events on the queue, or delete events from the queue. When all changes have been made, "Begin Execution" can be selected from the window. Execution then begins and the results are displayed in the EMACS window.

3.3 Testing

The purpose of this research effort was to enhance the aesthetic appearance and usability of the graphical interface to Architect without compromising or degrading AVSI's basic functionality. The formal testing of AVSI II took on two distinct aspects. The first aspect was a strictly objective test of AVSI II's functionality. All of this testing of AVSI II was conducted with applications developed from the digital logic-circuits domain, since that was the only fully validated domain available. The second aspect was an objective and subjective evaluation of the ability of an application specialist to use and understand the interface. AVSI II also received extensive informal testing. Two research efforts, concurrent with AVSI II development, were developing additional application domains for Architect. The domains were implemented through AVSI II, and the researchers provided valuable feedback and recommendations regarding AVSI II's appearance and usability. The details of the formal and informal testing and the test results are discussed in Chapter V.

3.3.1 Functionality Testing. This phase of the testing proved two points: AVSI II is fully backward compatible with all applications built and saved using either the Architect command-line interface or AVSI, and AVSI II did not negatively impact the functionality of Architect. First, four applications built prior to AVSI II development were parsed into AVSI II. They were each viewed in the edit-application-window, the update-algorithm-window, and the import-export-window to verify that the appropriate visual information was created to represent each application. The applications were executed and the behavior was verified against predicted behavior derived from a manual analysis of the circuit's design. Each application was saved back to system files and the new files were compared to the files of the originally saved applications to verify the integrity of the information. Secondly, four applications of varying complexity were built, executed, and saved. The execution results of each were compared to predicted execution results to verify proper behavior. Each saved application was parsed back into AVSI II and viewed in the edit-application-window, the update-algorithm-window, and the import-export-window to verify the information was properly re-created. Next, the applications were then executed to verify that proper behavior was preserved.

3.3.2 Usability Testing. Objective evaluation of AVSI II usability required development of metrics or figures of merit. Several figures of merit were considered. The time required to build an application and a measure of the errors made during application composition were considered and rejected as metrics. Time is highly dependent on environmental considerations like distractions and noise. Error rate, although normally a useful metric, was considered less important than the capability to or ease of recovering from the error. Additionally, the current users of AVSI II are highly experienced with the system and are not prone to errors. For these reasons, the time metric was not used and the error metric was monitored as part of the subjective testing discussed below. One useful figure of merit was developed. The measure counted the number of distinct user actions required to build an application. A user action is a specific activity such as clicking a mouse button on a part of the window or typing in a name from the keyboard. This measure positively reflected the reduction in user workload with AVSI II. To some extent, it also reflected the time required to compose the application. Data was collected for three different applications of varying complexity, each of which was built in AVSI and AVSI II.

This phase of the testing att r pted to measure how "good" AVSI II is. Much of the information collected here was derived subjectively by feedback from other members of the KBSE research group who were actively using AVSI II. Many comments on the interface's usability were collected informally as a consequence of the research group's close working proximity. Other information was collected more formally through a survey sheet that delineated the changes from AVSI to AVSI II and provided room for an evaluation of the form "significant improvement," "some improvement," "no improvement," "some degradation," or "significant degradation" of the change. Additional space on the form was provided for the user to make any appropriate comments. The feedback was consistently positive, and is discussed in more detail in Section 5.3.3.

3.3.3 Informal Testing. Two other KBSE researchers made extensive use of AVSI II during its design and development. Warner implemented the Digital Signal Processing (DSP) domain in Architect (32). His goal was to verify that a more substantial and complex domain than logic-circuits could be implemented in Architect. Waggoner actually implemented two domains: a time driven extension of the logic-circuits domain and a moving vehicle (a cruise missile) domain (31). His goal was to model and integrate time-dependent behavior in Architect.

Once their theoretical research and domain modelling was completed, each researcher integrated the domains into Architect and AVSI II. Since their efforts paralleled the development of AVSI II, they had first-hand experience with the capabilities and limitations of AVSI and AVSI II. Many comments and recommendations from their integration efforts contributed to the successful testing of AVSI II, and those comments are summarized in Section 5.3.3.

3.4 Conclusion

AVSI, the Architect Visual System Interface created by Weide (33), was a fundamental keystone of Architect's usefulness. Whereas Weide's research and implementation concentrated on functionality, this thesis effort concentrated on the appearance and usability of the interface. AVSI II allows the application specialist to perform the task of modeling and executing an application with less effort, in fewer steps, and in a more logical manner than he could with AVSI.

The next chapter provides information relevant to the design decisions and tradeoffs experienced in developing AVSI II, as well as the rationale behind those decisions. Appendix B provides a description of the details of how a domain modeler can construct an application domain's icon metaphors for use in AVSI II. Appendix A provides a sample session of creating an application with AVSI II.

IV. Design and Implementation of AVSI II

4.1 Introduction

A basic familiarity with INTERVISTA is all that is required to realize that its graphical power was inadequate for the development of AVSI II. As mentioned in Section 2.5, INTERVISTA's native window system is Common Windows. The purpose of Common Windows is to provide an interface from Lisp to the general window system located on the computer system. The version of Common Windows currently available uses the X window system and Common Lisp X Interface (CLX) (10:1-3). CLX, the standard X interface used by Common Lisp programmers, is a set of data types, functions, and macros that enable a Lisp user to interact with X (27:1-3). Common Windows (CW) and CLX (CW's underlying system) provide a wide spectrum of graphical capability. Careful review of Common Windows' capabilities indicated it would provide more than enough graphical augmentation to INTERVISTA to fulfill the goals of this thesis. Additionally, since direct calls to CW from within INTERVISTA are possible without translation, the integration of CW calls into AVSI II was straightforward. The relationship between the layers of software separating the application specialist from the hardware implementing AVSI II is depicted in Figure 4.1.

This chapter discusses the AVSI II design and implementation, highlighting Common Windows' role in the graphical enhancement and the reason behind the usability enhancements.

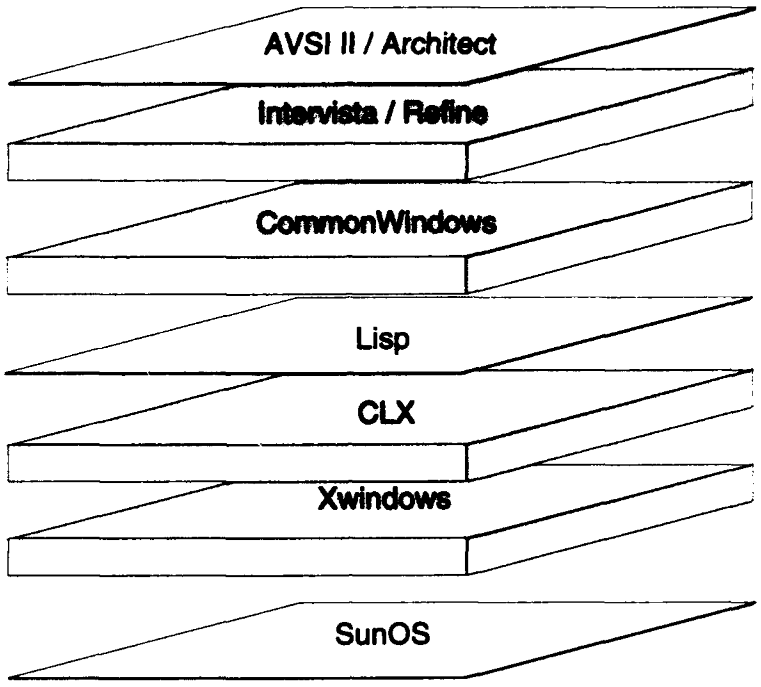


Figure 4.1 Software Layers Beneath AVSI

4.2 Design and Implementation Approach

AVSI II was developed using incremental development and rapid prototyping. Many of the basic concepts and procedures used to improve AVSI II were developed in design environments separate from Architect and AVSI. The first step involved creating an INTERVISTA window for design experimentation. Within the window various mouse handler, icon and link creation and manipulation, and window sizing options were investigated. Then, another experimentation window was developed in Common Windows. In that window the functions to manipulate window shape and size, window foreground and background colors, bitmap creation, bitmap foreground and background colors, text creation, creation of mouse sensitive areas, and primitive drawing commands were investigated. The knowledge from the Common Windows experiments was then applied to the INTERVISTA environment where the syntax for making Common Windows function calls within INTERVISTA functions was developed. This phase of the experimentation developed the knowledge required to display a bitmap from INTERVISTA; tie the bitmap to an INTERVISTA icon; create, move, and delete the icon-bitmap pair; and display colors.

Using the experimental windows to develop a design knowledge base made integrating the bitmap and color features into AVSI II a significantly easier task. Still, in order to maintain a fully functional interface during all stages of the research, an incremental development approach was adopted. The order of discussion in Chapter III is the order in which the interface was enhanced. Bitmaps were integrated first, then the update algorithm process was modified, and then the import-export connection process was redesigned.

4.3 Domain Specific Icons

The topic of domain-specific graphics was at the heart of enhancing AVSI. INTERVISTA is currently unable to provide icons other than box, ellipse, and diamond. Text icons are not considered real icons for the purpose of this discussion. Although the line thickness and icon height-to-width ratio can be controlled, the basic limitations still exist. Common Windows is capable of reading and displaying bitmap files in either X11 bitmap format or Interlisp-d format. The bitmaps can be viewed in an INTERVISTA window in conjunction with, or instead of, the INTERVISTA icon. A brief explanation of the icon formats follows:

1. The Interlisp-d format is actually a Lisp list that is converted into a bitmap stream by the CW `expr-to-bitmap` command. The most convenient way to build an Interlisp-d

bitmap is to use the CW drawing functions to draw the bitmap into a bitmap-stream and then save it to a bitmap file with `save-bitmap`.

2. The X11 bitmap format is significantly easier for the domain modeller to manipulate. X11 bitmap is the format for virtually all icons displayed on the screen of any system running the X windows system. Constructing the bitmap is easily accomplished using a paint or draw package that can save in X11 format, or using the OpenWindows^{TM} utility Icon Editor.

Since domain modelers should not be required to learn low-level drawing functions in CW, the X11 bitmap format was chosen. The bitmaps used in AVSI II were all drawn with Icon Editor, except for the subsystem bitmap used in the import-export-window. That bitmap was built in IslandPaint^{TM}, a graphics package capable of reading and writing X11 format files, and exported in X11 format. The following subsections discuss how the domain-specific icons for the AVSI II validation domain (i.e., the logic-circuits domain) were developed and detail the steps taken to integrate and display domain-specific icons in AVSI II.

4.3.1 Metaphor Development. Architect is used to instantiate and manipulate three classes of objects: Application-objects, Subsystem-objects, and Primitive-objects. Application-objects, subsystem-objects, and primitive-objects are defined in Architect's grammar and domain model [see (11) for a discussion of Architect's architecture]. A grammar defines the language used to characterize the objects in a domain, while a domain model defines the relationships and behaviors of the objects in a domain. While primitive-objects are generally defined in the Architect grammar and domain models, the relationships and behaviors of the primitives that are unique to a specific application domain are defined in a domain-specific grammar and domain model.

Because application-objects and subsystem-objects are structurally and functionally identical in the Architect environment regardless of the application being composed, the application- and subsystem-objects were each given a single metaphor (icon). However, the primitive-objects in a domain, although structurally similar, vary in functionality both between and within specific domains. The goal of a metaphor (or icon) is to reflect the behavior or function of the primitive-object, so individual metaphors for each primitive must be developed.

An application-object in Architect is the top-level, abstract entity that defines an application and its function. The metaphor for the application-object must convey a sense of regulation or supervision of its subcomponents. Figure 3.3 showed the icons developed to represent the application-object. Although the metaphor may not be readily apparent, the metaphor for an Architect application-object is an architect at a drafting table. The architect is the individual in charge of the design of a project; and he must be aware of, and in control of all aspects of the project's (or application's) design. The half-size icon (see Section 4.3.4 below) represents the top of the architect's drafting table where rolled-up plans or design tools would be seen.

The subsystem-object presented a significant challenge in metaphor design. Although the structure and functionality of a subsystem is a constant in Architect, subsystem meaning and relevance can be quite diverse between domains. The subsystem icon has one proverbial foot in the domain-specific arena and the other in the Architect general arena.

The final metaphor design is reflected in Figure 3.3. The icon looks like the outer edge of a partially completed puzzle. The metaphor is intended to convey the sense that a subsystem is a collection of parts – specifically, the controlled objects, or primitives, that comprise the subsystem. The fact that the puzzle is partially complete is significant in that an Architect subsystem can be a generic object, a saved but incomplete subsystem, a complete subsystem, or a subsystem still being defined.

Applying the metaphor design guidelines outlined in Chapter II was a straightforward task for the logic-circuits primitive-objects in the AVSI II validation domain. The following twelve primitive-objects are defined for the domain: and-gate, nand-gate, or-gate, nor-gate, not-gate, switch, led, multiplexer, counter, decoder, j-k flip-flop, and half adder. Each primitive is well defined and has a common visualization in the field of digital logic. The specific metaphor used for each primitive was extracted from the visual representations used in *Digital Logic and Computer Design* by Mano (17). Figure 3.6 shows the bitmaps drawn for the logic-circuits primitive-object metaphors.

The final step in metaphor design, according to Erickson (9:68-72), was to evaluate the effectiveness of the metaphors. Feedback from KBSE group members using AVSI II indicated a dissatisfaction with the switch icon. The original choice of a switch metaphor, although easily identified as a switch, implied an in-line switch requiring an input and an output. In fact, the switch implemented in the circuits domain only has an output. The inaccurate metaphor caused confusion concerning the actual functionality of the switch. Therefore, the metapho~ was redesigned to reflect the switch's true functionality. The final design reflects a single output toggle switch. The two designs are shown in Figure 4.2.

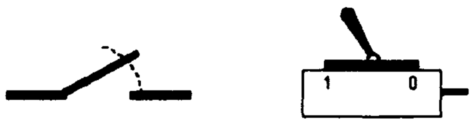


Figure 4.2 Switch Icons

Appendix B explains in detail the process a domain engineer should follow to develop a set of icons for use in AVSI II. The appendix discusses the Icon Editor utility, the size and format of the bitmaps to be created, the file naming convention used, and how to include the bitmaps in the domain model.

4.3.2 Icon Display. Once the appropriate metaphors were developed and the bitmap files were built using Icon Editor, the bitmaps had to be displayed in an INTERVISTA window. The icons displayed and manipulated in INTERVISTA are actually box icons with a line thickness of 0. The images seen on the screen are X11 bitmaps that are defined as attributes of the icon and are displayed at the same window location as the icon. INTERVISTA is not capable of displaying or manipulating a bitmap by itself, so the bitmaps are displayed on the CW native window that underlays the INTERVISTA window. To use an analogy, a CW native window may be perceived as a sheet of paper and the INTERVISTA window as a sheet of clear transparency film over the paper. In normal INTERVISTA operations the user can create and manipulate an icon on the transparency. AVSI II displays the bitmap on the piece of paper (the CW native window) at the same position as the INTERVISTA icon. The overall effect is an icon-bitmap pair that renders the appearance of a mouse sensitive and manipulatable icon in INTERVISTA.

An INTERVISTA window has a predefined attribute, ***native-window***, that holds the value of the underlying CW window. A bitmap is drawn onto the ***native-window*** using the CW command **`bitblt`**. **`Bitblt`** takes six arguments: the bitmap-stream to be drawn (created with **`read-bitmap`** as mentioned above), the x coordinate of the origin of the bitmap, the y coordinate of the origin of the bitmap, the native window to draw in, the x coordinate of the point to draw the bitmap, and the y coordinate of the point to draw the bitmap. An INTERVISTA icon's origin is defined as the center of the icon. A CW bitmap's origin is defined as the lower left corner of the bitmap. The bitmap had to be offset from the icon position by half the x and y dimensions of the bitmap in order to line them up. CW displays the bitmap by turning on or off the appropriate window pixels that form the image on the screen. Consequently, the bitmap is not a persistent artifact and must be redrawn whenever a change or update to the INTERVISTA window is made. The next subsection discusses the steps taken to make the bitmaps appear persistent.

4.3.3 Refresh and Redraw.

Displaying the bitmaps in INTERVISTA was only the first step in putting icon-bitmaps into AVSI II. The INTERVISTA manual specifically mentions the complication involved with going outside INTERVISTA to Common Windows.

> . . .if the contents of a window needs to be refreshed . . . , then INTERVISTA will not know to redraw any images that were drawn in a native window by Common Windows drawing primitives. You will have to explicitly specify the re-drawing operations in a function that is added . . . . (13)

Several functions were required to make the bitmaps appear persistent and interactive. The first obvious requirement is stated in the quotation above. When INTERVISTA refreshes a window, it erases the window and re-draws everything in it. However, INTERVISTA does not know how to execute a `bitblt` to redraw the bitmaps in the *native-window*.

The solution implemented involved writing a new function that would enumerate over all the icons in a window, accessing the bitmap attribute for each icon, and `bitblt`-ing the bitmap in the proper location. The function, called **redraw-icons**, was added to INTERVISTA window's default refresh function with a form similar to the one shown here:

```
form ADD-MY-REDRAW
   let (old-redraw-functions = redraw-functions(*SS-WINDOW*))
       (redraw-functions(*SS-WINDOW*) <-  ['redraw-icons, $old-redraw-functions])
```

The second issue involved moving icons in the windows. INTERVISTA does not perform a window refresh as a consequence of moving an icon. Consequently, moving an icon resulted in the appearance of the icon at the new position with its proper bitmap displayed, and a bitmap still visible at the icon's original position. The fix was straightforward and involved replacing the default function, `user-move-icon`, with a call to a new function that performed `user-move-icon` and then refreshed the window.

The last display issue involved deleting an icon. Deleting the icon with the INTERVISTA `delete-icon` function does not refresh the window, so the bitmap for the icon persisted in the *native-window* after the icon was deleted. The fix was the same as for moving an icon. A new function was written that called `delete-icon` and refreshed the window.

4.3.4 Icon Scalability. To make an INTERVISTA window and its contents visible on the screen, the surface viewed by the window must be defined with **view-surface**. One action of **view-surface** is to set the window's attributes so that the information in the window is centered (*surface-origin*) and scaled (*scale-factor*) so that everything in the window is visible. While this is satisfactory for icons, the bitmaps drawn by CW are static objects that cannot be easily scaled. Although an algorithm to dynamically draw (or scale) every bitmap using the CW primitive drawing functions could have been developed, that solution would have resulted in severe time penalties for redrawing screens with many icons. It would also have necessitated that the domain modelers be able to write drawing and scaling algorithms for every primitive-object instead of building the static bitmaps in Icon Edit.

The method implemented for AVSI II was to restrict the application specialist to two menu selectable icon-bitmap sizes. The sizes are full-size (100%) and half-size (50%). The domain modeler is responsible for two bitmaps per primitive-object. This is considered to be a reasonable task. The specific size and format requirements for the bitmaps are discussed in Appendix B. Selecting "Rescale Window" from the window menu calls the `change-scale-factor` function that sets the window's *scale-factor* to 1.0 for full-size or 2.0 for half-size. Since the INTERVISTA windows have scroll bars and a reshape function, the application specialist can scale, scroll, or reshape as necessary to view an entire application.

4.3.5 Required Attributes. In order to properly display an icon-bitmap pair in an INTERVISTA window, the icon must have certain attributes:

- clip-icon-label?: Pre-defined in INTERVISTA, this attribute controls whether an icon's label is completely displayed or truncated at the edge of the icon. Since the icon itself is not visible in AVSI II, this attribute should be set to false.
- border-thickness: Pre-defined in INTERVISTA, this attribute controls the thickness of the edge of the icon. To make the icon "invisible" this attribute needs to be set to 0.

- bitmap4icon-l: An AVSI II-defined attribute, it stores the bitmap stream for the full-size bitmap. The attribute is defined as a map between an icon and an any-type.
- bitmap4icon-s: An AVSI II-defined attribute, it stores the bitmap stream for the half-size bitmap. The attribute is defined as a map between an icon and an any-type.

Although there are many other attributes of an icon, the default values are sufficient for AVSI II. These attributes are set in the primitive-object's visual description file for each domain and parsed into AVSI II at runtime. The same attributes are internally set for the application-object and subsystem-object icons.

4.4 Update Algorithm

Four changes were made to AVSI in the update algorithm creation windows. Two of them were cosmetic changes and two were functional. The cosmetic changes included adding color to the background of the windows and closing all the subwindows simultaneously. Color was added to the controllee window and update algorithm window using the process described in Section 4.7.1. Closing all update windows simultaneously involved replacing the default "Deactivate" menu command of `window-active?(w) <- false` with a function call that executes :

```
true --> (window-active?(*algorithm-window*) <- false;
          window-active?(*controllee-window*) <- false;
          window-active?(w) <- false)
```

The two functional changes are a new controllee-window and the process by which the update algorithm is created. During the application composition, subsystems are linked to the application and subsystems and primitives are linked to other subsystems. The linked objects are called the "controllees" of the controlling application or subsystem. The existence and reuse of the set of controllees for an application or subsystem simplified the update algorithm definition.

4.4.1 Controllee Window. The controllee-window is created in the same fashion as all other diagram windows in AVSI II. To make room for it on the screen, the textual update algorithm window was moved to the right edge of the screen, and the controllee-window was placed in the resulting gap. The **`display-controllees`** function creates the window, accesses the set of controllees for the current subsystem or application, and displays the controllees in a grid pattern in the window. The function then creates a pair of icon-bitmaps to represent if-then and while-do statements and displays them in the window.

4.4.2 Update Object Creation. Once the controllee-window is populated, the update algorithm can be built. When the mouse is clicked on a controllee, the INTERVISTA function **`get-link`** is called. **`Get-link`** prompts the application to select a link by clicking the mouse on a link's nub. Once the link is selected, a copy of the controllee's icon-bitmap is inserted in the update sequence, the icon-bitmap is linked to the icon-bitmaps immediately preceding and following it in the sequence, the originally selected link is deleted, and the textual update sequence is built as it was in AVSI. Once the update algorithm reaches seven icon-bitmaps in length (including the start and end icons), the *scale-factor* of the window is automatically set to 2.0 for half-size.

4.5 Import and Export Areas

The AVSI II changes to the import-export connection process were all additions to the prior AVSI capabilities. Only one function was deleted; some were modified; and several were added. AVSI treated the subsystem and primitive icon groups as individual icons and required a special function to move them as a group. AVSI II defines the import-export icon set as a supericon with a set of subicons. When a supericon is moved with **`user-move-icon`** the subicons are automatically moved to the appropriate relative positions from the supericon. As a result, the AVSI function **`move-icon-group`** became unnecessary and was deleted. The modifications and additions are discussed below.

4.5.1 Import - Export Icon Creation. The creation of a subsystem icon is unchanged from AVSI with the exception of using a sub/super-icon group instead of the informal icon grouping. The primitive-object icon creation is somewhat more complicated. The supericon is created by accessing the controllees of the parent subsystem or application in the same fashion as the controllee-window for the update algorithm. For each primitive in the controllee set, the domain model is accessed using a **`get-input-output-variable`** function. The function returns the set of inputs or outputs each primitive has defined for it. The input and output sets are passed, along with the supericon, to a **`make-ie-subicons`** function. **`Make-ie-subicons`** creates a subicon for every input and output, labels them appropriately, evenly distributes the input subicons along the left edge of the supericon, evenly distributes the output subicons along the right edge of the supericon, creates the subicon-supericon relationships, and returns the icon set for display.

The method of populating the import-export window is the same as populating the controllee-window. The main difference is icon-bitmap position. If the attribute *comp-position* of the object corresponding to the icon-bitmap is defined, the icon-bitmap takes that value as its position in the window. If the value is undefined, the icon-bitmap is passed to the `make-lattice3` function that places the icon-bitmap and defines the *comp-position* value. Once the window is populated with the controllees, the connection process can begin.

4.5.2 Primitive Icon Connection. The connections between primitives was greatly simplified for AVSI II. To connect a specific output on a primitive to a specific input on a primitive, the application specialist simply clicks on the output and then the input. A visible link is created with `link-exp-to-imp` and the import-export connection is made. The mouse handlers allow the connection to be made in the reverse order using `link-imp-to-exp`. When the output is selected, its name is loaded into the variable *CURRENT-EXPORT*. When an input is selected, the function checks *CURRENT-EXPORT* and, if it is defined, a connection is made. If *CURRENT-EXPORT* has not been defined, the input is loaded into *CURRENT-IMPORT*, and, when an output is selected, the connection is made. If the parent subsystem's MSP-window(s) are visible when the connection is made, the MSP-window(s) are updated. MSP-windows are Mouse Sensitive Printing windows consisting of mouse sensitive text entries that correspond to objects in the REFINE object base. In this instance, the entries correspond to import and export objects.

The subicon's highlight state is used to feed back to the application specialist if and where the import export is connected. The three states (discussed in Section 3.2.4) are as follows:

- Highlighted - The icon is displayed in reverse video by setting the icon attribute *highlight-style* to 'reverse-video. This state represents an unconnected subicon.
- Unhighlighted/Invisible - The icon is not visible on the screen. This state is displayed by setting the icon attribute *highlight-style* to 'unhighlighted. The state represents an icon that is connected to another icon visible in the current window.
- Thin empty box - The icon is displayed as a thin-walled, empty box. This state is displayed by setting the icon attribute *highlight-style* to 'unhighlighted and the attribute *border-thickness* to 1. The state represents an icon that is connected to another icon not visible in the current window.

If the application specialist clicks the mouse on a link between icons, he is given the option to delete or redraw the link. If delete is selected, the visible link is erased from the window, the import-export connection in the object is erased, and the MSP-window(s) are updated. If the application specialist selects the redraw option, the function `user-layout-link-path` is called and the link path can be manually drawn. The function changes the mouse cursor into an INTERVISTA "pencil," and the application specialist can draw the path by moving the pencil to a sequence of points on the window and clicking the mouse button at each point (see Figure 4.3). The link path is completely drawn when the mouse is clicked over an icon.

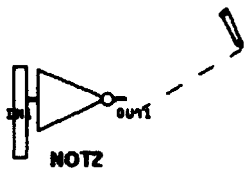


Figure 4.3 INTERVISTA's Link Pencil

4.5.3 Inter-Subsystem Communication. Connecting imports and exports across subsystems is unchanged from AVSI's implementation. The only difference is the icon-bitmap used for a subsystem. The bitmap was designed to look like an Object Connection Update (OCU) import area, export area, controller group. The bitmap was drawn in IslandPaint and saved in X11 bitmap format. The bitmap is displayed and updated in the window with the `bitblt` command as mentioned in Section 4.3. The icons are a supericon (controller) with two subicons.

4.5.4 AVSI Compatibility. The mouse handler and connection routines were written such that the AVSI process of making connections from the MSP-windows is fully interchangeable with the new AVSI II process. By defining the *CURRENT-IMPORT* and *CURRENT-EXPORT* variables as global to this process, the application specialist can connect appropriate text entries (objects) in the MSP-window with subicons in the red import-export-window and vice versa. Additionally, any addition or deletion made in one window is immediately reflected in all other visible windows. This entire process was made possible by replicating the code to connect objects in MSP-windows for the subicon connections and hooking it all together with functional converses.

4.6 Application Executive

Welgan's research into providing an Application Executive capability for Architect (34) required the addition of menus and MSP-windows to AVSI II. The Application Executive, among other capabilities not relevant to this discussion, allows the application specialist to choose execution modes other than simple sequential updates. The changes made to support the executive are discussed below.

4.6.1 Execution Mode Support. To allow Architect to execute in the proper mode, the application specialist must first select the desired execution mode when the application is created. This was supported by adding a single-menu in the `create-new-application` function to display the available execution modes and prompt for a choice. The selected item is stored as an attribute of the application.

Selection of an event-driven mode obviates the requirement to build update algorithms. The update algorithm contains the sequence in which the primitive objects in the application must update their state information so that data is propagated through the application to the output in an orderly and accurate manner. The event-driven execution mode is not concerned with data propagation; it is concerned with event generation and processing. To prevent the application specialist from wasting time building an update algorithm that will not be used, whenever the application specialist chooses "Edit Application Update" or clicks on a subsystem's controller icon, AVSI II accesses the application's execution mode attribute and, if it is an event-driven variant, the update algorithm function call is aborted and a "This Process is Not Required for This Application" message is displayed in the AVSI II control window.

4.6.2 Event Queue Support. For an event-driven execution mode, the application specialist needs to have the option to pre-load the event queue, or modify or delete items from it. When the application specialist selects "Execute The Application," the event queue is displayed in an MSP-window. The window also displays the options to "ADD," "MODIFY, " or "DELETE" queue entries as described in Section 3.2.5. Adding an event is accomplished by selecting a new event-object type from a pop-up list of types and populating the attributes of the object through a series of `get-string` windows. When all attributes are completed, the function **add-event** is called to insert the event-object in the proper location in the queue. Modifying an event on the queue is accomplished by selecting the event to modify, deleting the selected event, and adding a new one of the same type. Deletion is accomplished by simply locating the event-object in the queue (represented by a sequence of event-objects) and removing it from the queue.

4.7 Miscellaneous

4.7.1 Color in AVSI II. Since the current version of INTERVISTA does not provide color capability, function calls to CW were required to integrate color into AVSI II. Within CW, windows and bitmaps are defined as two-color objects (i.e., foreground and background color). CW has eight pre-defined colors available: red, green, blue, magenta, yellow, cyan, black, and white. Additionally, CW can access any named X11 color or create a new color using the `make-color` command. Chapter III discussed the uses of color for the background in certain windows. The color was loaded using the CW/Lisp command `setf( cw::window-stream-background-color(native-window(*ss-window*)),`*color*) inside a form. As with the other CW commands, this one changes the color on the *native-window*, not the INTERVISTA window. The difference, however, is that the color in a window is persistent because INTERVISTA's refresh function does not affect the background, only the foreground of a window. Because bitmaps are drawn in the foreground, a window refresh affects them.

There is one exception to the statement in Section 3.2.2 which mentions that color was not used for the bitmaps. While bitmap-foreground color was not changed from black (the default), the background had to be changed to match the window's background. The default bitmap-background color is white, and the consequence of not matching backgrounds is an icon in a small field of white on a colored window. The bitmap color had to be changed in **`redraw-icons`** function using a command like **`setf( (cw::bitmap-background-color( bitmap4icon-l(x))), bcolor)`**, where *bcolor* is the background color of the window being refreshed.

4.7.2 Multiple Link Function. The **`make-all-links`** function allows the application specialist to link all the primitive- and sub-subsystem-objects to the current subsystem-object in one step. Construction of the AVSI II **`make-all-links`** function was straightforward. When a subsystem- or application-object's icon is mouse selected and **`make-all-links`** is selected, the function first copies the *icons(surface-viewed(diagram-window))* set to a new set. Next all icons that are already connected to another icon are deleted from the set. The icons left in the set are displayed to the application specialist in a multiple-menu. A multiple-menu allows the user to select all, some, or none of the entries in the list. The icons selected from the menu are stored in another set that is enumerated over. Each is automatically linked to the originally selected icon with the same routine used in AVSI to make a single link.

4.7.3 Parse and Save Functions. The desire to retain the icon-bitmap window positioning in the import-export window required a change to both the Architect architectural model and the domain-specific architectural model. Two new attributes of each component object were created: *comp-x* records the x-coordinate of the icon as a map between the component-object and an integer, and *comp-y* records the y-coordinate of the icon. A new attribute of all component-objects was also added to AVSI II. The attribute, called *comp-position*, is a map from a component to the INTERVISTA type POINT. The function `get-position` converts *comp-x* and *comp-y* into *comp-position* when a saved application is parsed into AVSI II. Conversely, when an application definition is saved to a file, the function *retrieve-position* accesses *comp-position* and converts the POINT value to *comp-x* and *comp-y*.

4.8 Summary

This chapter outlined the decisions motivating the design of AVSI II and the details of its implementation. The goals of AVSI II's design were to enhance the visual presentation and interface usability for an application specialist building an application with Architect while preserving the AVSI functionality. The next chapter describes how AVSI II was tested, results of that testing, and an analysis of the results.

V. Testing and Validation of AVSI II

Validation of AVSI II encompassed testing of two general categories. The compatibility and functionality of AVSI II were verified by using the interface with applications constructed prior to AVSI II development. The usability of the interface was tested via an objective analysis of the reduction in workload realized through AVSI II and with subjective evaluations by members of the KBSE research group. This chapter reviews the testing of AVSI II, the results of the testing, and an analysis of the test results. Additionally, an analysis of the strengths and weaknesses of AVSI II is provided.

5.1 Validation Domain

AVSI II was formally tested using applications developed in the digital logic-circuits domain. The interface was informally tested throughout its development by other KBSE researchers who were using the interface during their research. Since the domain was previously validated (1, 22, 33), there was no requirement to exercise the operation of each domain primitive; therefore, test applications were designed to verify the interface's capability, not the domain's.

The test applications were designed for and tested in sequential mode, non-time-driven execution. This mode is compatible with AVSI's capability and, subsequently, resulted in a valid comparison of the two interfaces. This test philosophy did not provide any indication of the proper function of time-based or event-driven execution modes of Architect; however, AVSI II does not need to provide any functionality over and above mode selection and queue manipulation to support those modes.

5.2 *Compatibility Testing*

With the implementation of specific icon-bitmaps for the domain primitives, there was a need to verify that the new and modified visual attributes would be properly created for applications developed prior to AVSI II. Four applications that were created and saved during previous research were used. Each application was loaded, in turn, and viewed in the edit-application-window, the edit-subsystem-windows, the update-algorithm-windows, and the import-export-windows. The applications were executed to verify their behavior, and they were then re-saved. Next, the saved files were compared to the original files for consistency.

Only one anomaly was discovered. An application saved by Architect does not contain any visual information. The function that creates an icon for an object and defines it as a functional converse for the object is executed while the blue edit-application-window or edit-subsystem-window is populated. Therefore, although the application load function populates the object base, it does not create the icons and converses. This requires that the edit-application-window be exposed, and thus the icons and converses be defined, before the update-algorithm-windows or import-export-windows can be viewed. Otherwise, the icons required for display in those windows would not have been created yet. It is important to note that since visual support is not provided for execution, the loaded application can be executed without having to expose the edit-application-window first. This anomaly was an unforeseen side-effect of integrating domain-specific graphics into the update algorithm and import-export windows, and could be corrected by replicating the icon creation function calls in the application loading routine.

Four additional test applications were created with AVSI II and subsequently executed, saved, and reloaded. The loaded applications looked and executed precisely like the originals.

5.3 Usability Testing

An objective evaluation of AVSI II's impact on the size of the application specialist's composition effort was measured for four applications. The data collected and an analysis of the results are presented in the following subsection. The next subsection presents the comments and evaluations collected from the other KBSE research group members. Two group members used AVSI II extensively during their development of new application domains for Architect (refer to Section 3.3.3).

5.3.1 User Workload. The metric developed to test AVSI II's usability was the number of user actions required to build an application. Four applications of varying complexity were built with both AVSI and AVSI II. The number of user actions required to proceed from application creation to execution was counted. A user action was defined as an atomic effort (like a single click of a mouse button) or inputting a value (like an icon label). Two sets of data were collected for AVSI II. The first value, listed under the header "AVSI II," assumed the application specialist was unconcerned with icon position in the import-export-windows. These values represented a more accurate comparison with AVSI since AVSI did not provide for icon positioning. The second set of data, listed under the header "AVSI II+," assumed the application specialist moved every icon in the import-

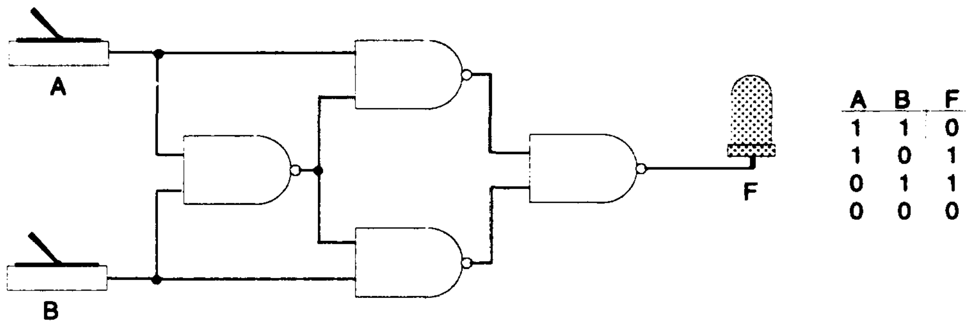


Figure 5.1 Exclusive-OR Circuit and Truth Table

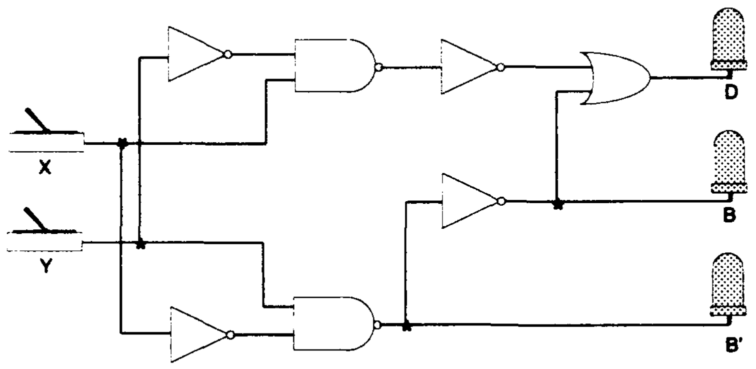
Table 5.1 Exclusive-OR Test Results

	AVSI II	AVSI II+	AVSI
Create/Edit Application	15	15	15
Application Update	7	7	14
Create Subsystems	43	43	62
Subsystem Updates	22	22	66
Build Imports-Exports	36	60	43
Execute	1	1	1
TOTAL	124	148	201

export-window to new positions. These values represented a truer picture of AVSI II's user workload because most users will exercise a feature if it is available.

5.3.1.1 Exclusive-OR. The first test circuit developed was a four Nand-gate implementation of an Exclusive-OR gate. The component layout and truth table for the circuit are shown in Figure 5.1. Table 5.1 shows, by development area and total, the number of user actions required to build the application.

The data indicate a 39.3% reduction in workload from AVSI to AVSI II without icon repositioning, and a 26.4% reduction with icon repositioning in the import-export-window.



X	Y	D	B	B'
1	1	0	0	1
1	0	1	0	1
0	1	1	1	0
0	0	0	0	1

Figure 5.2 Half Subtracter Circuit and Truth Table

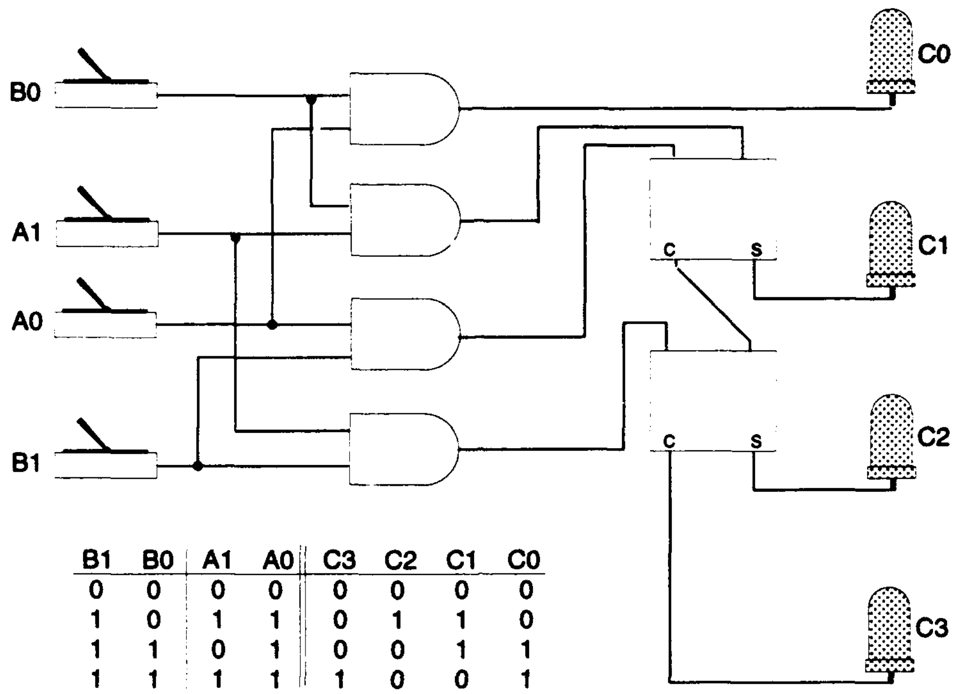
Table 5.2 Half Subtracter Test Results

	AVSI II	AVSI II+	AVSI
Create/Edit Application	15	15	15
Application Update	7	7	14
Create Subsystems	58	58	93
Subsystem Updates	32	32	101
Build Imports-Exports	44	83	61
Execute	1	1	1
TOTAL	157	196	285

5.3.1.2 Half Subtracter. The second test circuit developed was a seven gate implementation of a Half Subtracter. The component layout and truth-table for the circuit are shown in Figure 5.2. Table 5.2 shows, by development area and total, the number of user actions required to build the application.

The data indicate a 44.9% reduction in workload from AVSI to AVSI II without icon repositioning, and a 31.2% reduction with icon repositioning in the import-export-window.

5.3.1.3 Binary Array Multiplier. The third test circuit developed was a four gate, two half-adder implementation of a Binary Array Multiplier. The component layout



B1	B0	A1	A0	C3	C2	C1	C0
0	0	0	0	0	0	0	0
1	0	1	1	0	1	1	0
1	1	0	1	0	0	1	1
1	1	1	1	1	0	0	1

Figure 5.3 Binary Array Multiplier Circuit and Truth Table

and truth-table for the circuit are shown in Figure 5.3. Table 5.3 shows, by development area and total, the number of user actions required to build the application.

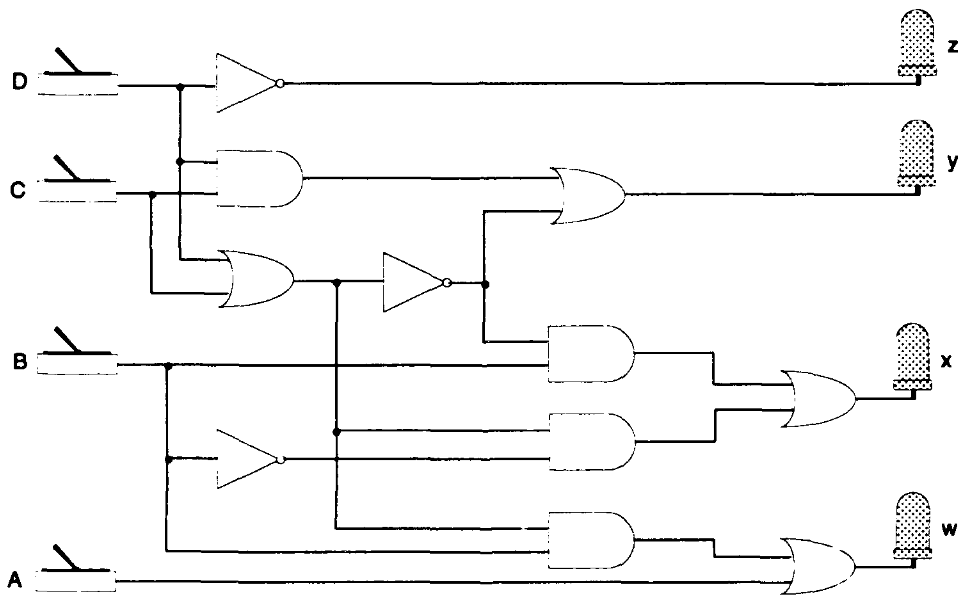
The data indicate a 45.0% reduction in workload from AVSI to AVSI II without icon repositioning, and a 30.1% reduction with icon repositioning in the import-export-window.

5.3.1.4 BCD to Excess-3 Decoder. The final test circuit developed was an eleven gate implementation of a BCD-to-Excess-3 Decoder. The component layout and truth table for the circuit are shown in Figure 5.4. Table 5.4 shows, by development area and total, the number of user actions required to build the application.

The data indicate a 46.9% reduction in workload from AVSI to AVSI II without icon repositioning, and a 31.7% reduction with icon repositioning in the import-export-window.

Table 5.3 Binary Array Multiplier Test Results

	AVSI II	AVSI II+	AVSI
Create/Edit Application	15	15	15
Application Update	7	7	14
Create Subsystems	61	61	102
Subsystem Updates	36	36	115
Build Imports-Exports	46	91	55
Execute	1	1	1
TOTAL	166	211	302



BCD				Excess-3 code			
A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

Figure 5.4 BCD to Excess-3 Decoder Circuit and Truth Table

Table 5.4 BCD to Excess-3 Decoder Test Results

	AVSI II	AVSI II+	AVSI
Create/Edit Application	15	15	15
Application Update	7	7	14
Create Subsystems	78	78	129
Subsystem Updates	46	46	150
Build Imports-Exports	62	122	85
Execute	1	1	1
TOTAL	209	269	394

5.3.2 Workload Analysis. The data shows a significant reduction in the application specialist's workload to build an application with AVSI II as opposed to AVSI. The average reduction for the four applications was 44.0% for applications where the user did not reposition icons in the import-export-windows, and 29.9% for applications where the user did reposition icons in the import-export-windows. The data supports a conclusion that the percent of reduction in effort increases as the number of objects in an application increase (at least for the option of nor repositioning icons). The largest improvement was realized in building the update algorithms. The AVSI implementation required seven distinct actions to add a single statement to the update algorithm. The AVSI II implementation required only two actions, a 71% reduction in effort. The other major improvement arose from the "Link Multiple Targets" feature in the edit-application-window and edit-subsystem-window. AVSI II allows the application specialist to link every subsystem and primitive to the top subsystem with four user actions as opposed to three user actions per subsystem and primitive with AVSI. Lastly, the process of making the import-export connections was not overly simplified from a workload standpoint. With the addition of the icon positioning capability, this area became more workload intensive. The option exists, however, to ignore icon positioning if position is not important.

5.3.3 User Evaluations. The data presented in this section were extracted from the comments AVSI II users and the responses to the survey on AVSI II enhancements. The informal comments were centrally recorded throughout the AVSI II development in response to comments and recommendations from those KBSE research members actively using the interface in their research. The surveys were distributed to the same researchers once AVSI II development was complete. The survey contained a set of questions concerning aspects of AVSI II's functionality and asked if the changes from AVSI represented an improvement or degradation in capability. Each question also had room for the researcher to add comments if desired. Listed below is a summary of the comments recorded during the interface's development and received on the survey forms.

- A grid in the import-export-window would be helpful in lining up the primitive icons.
- Feedback for possible import-export connections between primitives, similar to that provided between subsystems, would be a useful feature.
- Another icon highlight style in the import-export window is needed to show something is connected to both visible and non-visible objects.
- Closing multiple windows at once (the update windows and the import-export windows) is a real time-saver.
- I like the "Link Multiple Targets" option. I don't forget to link an object using the feature.

- The new update process is much easier to use. I didn't like having to remember the names of all my primitives.
- The colored windows are a good idea. I can tell which windows are open at a glance.
- The new icons make finding a primitive in the technology base window much easier.

One other comment was made by several different people. They felt that the colors used in the individual windows were irrelevant. The fact that there was color and that the colors were different for different functional areas was the significant issue.

The responses on the survey were consistent. The conclusions that can be drawn from the responses are:

1. The impression of the application and subsystem icons was basically neutral. Users felt that the design was an improvement over the AVSI simple-box icon, but the specific icon design was not anything special.
2. The `Link Multiple Targets` function was seen as a big time saver in the edit-subsystem-window.
3. The ability to close multiple windows from a single `Deactivate` command is a nice convenience.
4. The ability to rescale the icons in a window was seen as an improvement by most users, although one user felt it was a degradation from AVSI's automatic window scaling.

5. The domain-specific icons received the most enthusiastic feedback. The users felt that locating primitive-objects in the technology-base-window was much easier with the icons.

6. The redrawn subsystem diagram window received an almost unanimously neutral response.

7. The controllee-window by itself was not seen as a big improvement; however, using the controllee-window to populate the update algorithm sequence was a significant improvement.

8. The changes to the import-export connection process between primitives were perceived as an improvement. Using the domain-specific icons in the window was a significant improvement. Icon positioning was some improvement. The icon highlight styles were understandable; however, one user desired more than the three styles in order to display that a primitive was connected to both a visible object and a not-visible object.

The general impression of the users of AVSI II was that the visual enhancements to the interface were a greater improvement to the system than the usability enhancements were. The usability enhancements were a time savings, though; especially the `Link Multiple Targets` and redesigned update algorithm composition processes.

5.4 AVSI II's Shortcomings

The visual enhancements of AVSI II and the increased usability of the interface are significant accomplishments. However, there is always a new feature or a better layout for any display. The evolutionary development of Architect requires that the graphical interface also evolve to incorporate new functionality. The following areas of AVSI II still require significant research:

1. Support for the Architect Application Executive. AVSI II provides only rudimentary support for the power of the Executive. The application specialist must be given the capability to interface with and possibly configure the Application Executive function for Architect. Initially, Architect and AVSI II were designed to support a sequential update execution mode. Time-driven sequential execution modes and event-driven execution modes are under development to support a variety of domains. More application specialist control over the event queue before, during, and after execution will be required to support the new modes. For time-driven domains, a system clock will be required as part of the Application Executive. Additionally, the time relevant information should be displayed to the application specialist.

2. Visual support for semantic analysis. This area was identified as a shortcoming for AVSI; the deficiency persists into AVSI II. Semantic analysis is reflected in AVSI II as a simple "Semantic checks passed" message or a failure message with instructions to look at the Emacs window for details. The semantic functions of Architect should be strengthened to accurately report the specific semantic violation and AVSI II should have the capability to present the erroneous design area in a user editable window with the specific error highlighted.

3. Visual support for application execution. This was also identified as a shortcoming of AVSI. AVSI II was not intended to solve this shortcoming, but execution visualization

is a logical extension of the interface. Using the concepts developed for AVSI II, an object's state could be displayed using color, highlighting, or movement. Color and highlight style are maintained as attributes of an icon, and an icon is maintained as a functional converse of an object. The state of an object is simply an attribute or collection of attributes of the object. Therefore, it is a straight-forward task to monitor the state of an object by checking its attributes, and set or modify the values of the corresponding icon's color or highlight style attributes. The location of an icon in a window is stored as an attribute called *position* which is of type POINT. Movement of an icon can be simulated by repeatedly re-drawing the icon (and bitmap) at a sequence of positions. The speed at which the icon is redrawn and the distance between successive points will determine how smooth the movement appears.

5.5 Summary

The concepts applied to this research, and summarized in Chapter II, have shown that a sophisticated graphical interface can be developed to support a visual composition process. AVSI II was a positive step in enhancing the interface to Architect. The capability to display and manipulate domain-specific icons makes designing an application more intuitive for the application specialist. The bitmaps and colors increased the application specialists ability to find primitive-objects in the various windows and to immediately recognize in which window he is working. The revised processes for building the update algorithms for sequential execution applications and the expanded visualizations for the import-export connection processes reduce the user's workload and reduce the possibility of error from misnamed update objects or improper connections. AVSI II proved to be fully compatible with previously designed applications, thus preserving the efforts of earlier research. This chapter reviewed the process by which AVSI II was tested and validated. Also, it summarized the feedback on the interface's usability from users of the system. Some of AVSI II's shortcomings were mentioned. Those shortcomings, as well as some of the user feedback comments, are addressed in more detail in the next chapter.

VI. Conclusion and Recommendations

The original goals for developing the AVSI II visual interface, as discussed in Chapter I, were to enhance the graphical presentation and usability of the interface to Architect, a domain-oriented application composition and generation system. The existing graphical interface for Architect, AVSI, was developed in INTERVISTA and suffered from a limited range of graphical display options. AVSI II overcame the display limitations by augmenting INTERVISTA's graphics with additional functionality provided by Common Windows, a graphical development package for Lisp and the graphical foundation for INTERVISTA. The specific design goals stated in Chapter I are:

1. Develop a set of domain-specific graphical icons (metaphors) for an existing Architect application domain and integrate the graphics into AVSI II's application construction process.
2. Develop the means for domain engineers to develop and incorporate additional iconic metaphor sets for new application domains.
3. Redesign the update algorithm definition process to eliminate duplication of effort and tedium.
4. Redesign the import/export connection process by exploiting domain-specific graphics to make object connection more intuitive.
5. Support the visualization needs of concurrent research activities.
6. Test the new interface and quantify AVSI II's improvement in the areas of usability and composition effort reduction.

6.1 Results and Conclusions

AVSI II successfully achieved the original research goals. Design and development were conducted through rapid prototyping and incremental improvement. This philosophy proved to be highly successful. The Software Refinery^{TM} development environment of REFINE and INTERVISTA was well suited for prototyping based on its transformational capabilities. Each step in the research built upon earlier accomplishments and extended the capability of AVSI II. Achievement of the four goals can be summarized in three areas of activity:

6.1.1 Domain-Specific Graphics. Use of domain-specific bitmaps made specific primitive-objects in the technology-base-window readily identifiable. The application specialist does not have to rely on the object class names to differentiate between the primitives. The graphics also made locating specific objects in an application easier by first focusing the application specialist's attention on the type of primitive-object and then letting the icon label identify the specific object.

The design rules and concepts by which the metaphors and bitmaps for the logic-circuits domain were developed for AVSI II was generalized into a process that can be applied by any domain engineer or domain modeller to a new domain development. The process is delineated in Appendix B. Using the steps in Appendix B, Warner successfully developed a set of metaphors and bitmaps for the Digital Signal Processing (DSP) domain (32), and Waggoner developed a set of metaphors and bitmaps for the cruise missile domain (31).

Color backgrounds were integrated into several AVSI II windows. Color proved to be a powerful context cue for users of the interface. Color was not integrated into the design of individual icons in AVSI II, primarily because of the two color limitation of the X11 bitmap format used with Common Windows; however, Section 6.2 discusses some potential research in the area of icon color.

6.1.2 Update Algorithm. The integration of icon-bitmaps into AVSI II provided a significantly wider range of application design options. The greatest success in AVSI II's usability enhancement was realized in constructing the update algorithm for the application and its subsystems. Creation of the controllee-window alleviated the user from the need to remember the names of the objects being added to an update. The process of clicking on the controllee icon and "dropping" it in the update sequence significantly reduced the effort required to build the update algorithm.

6.1.3 Import-Export Areas. Integrating domain-specific graphics into the import-export connection process was a challenge in visualization. Constructing the import-export connections for primitives within a subsystem emulates the process of "hooking up" a system. This process fits a user's concept of reality and is easy to comprehend. The process of connecting objects in one import/export area to an object in other export/import areas remains an OCU-like abstraction and, therefore, is a straightforward concept to assimilate. However, connecting a primitive's inputs or outputs to imports or exports in the MSP-window of another subsystem does not fit any previous conceptual models, but the process is uncomplicated, and a user can easily adapt to it. There is a concern that the multiple possible ways to make connections (i.e., icon to MSP-window object, MSP-window object to icon, MSP-window object to MSP-window object, and icon to icon) within the same set of windows could present too many options for a user.

6.2 Recommendations for Further Research

1. *Explore color use or highlight styles in bitmaps.* The two color limitation of the Common Windows/X11 bitmap format and a desire to keep the bitmap development simple resulted in the use of only black icons in AVSI II. There are drawing capabilities in Common Windows that extend beyond simply "`bitblt`-ing" a bitmap to a window. Additionally, tapping the functionality of CLX or X Windows itself sh ·ld be explored.

   I. ERVISTA has the capability to accept user defined icon types. The process of defining an icon type effectively requires the programmer to write a mini-program of instructions to INTERVISTA detailing how the icon can be drawn and manipulated. Assuming an Architect domain-specific icon type can be developed, it would solve the problems of icon scalability that currently restrict AVSI II to either half-size or full-size. Additionally, INTERVISTA has an undocumented icon type defined as BITMAP. The BITMAP type is not supported in the current version of INTERVISTA; however, it is possible that future releases of the software will support the icon type.

   INTERVISTA currently supports two highlight styles: unhighlighted or reverse-video. New highlight styles can be defined in the same fashion that new icon styles can be defined. The use of grays or cross-hatches as alternative highlight styles would give the interface developer more flexibility in communicating with the user.

2. *Explore the use of sounds in the interface.* By making LISP calls to the host operating system, an interface developer can incorporate sounds into the interface design. Sounds are a common part of user interface design, and, although easy to misuse or abuse, sound can provide strong reinforcement or feedback to a user. Computers typically beep at a user if an error occurs, or chime upon successful completion of a task.

3. *Extend user feedback in the controllee-window.* The controllee-window in AVSI II provides no visual feedback as to which controllees have been incorporated into the update algorithm (other than visual inspection of the textual algorithm window). Although this is not a problem for a simple sequential update, when If-Then or While-Do statements are being constructed, the user may wish to know which controllees have or have not been used. One potential solution would be to change the color of the icon in the controllee-window once it has been used in the update somewhere, or to highlight the icon in some other fashion.

4. *Develop a structured process for If-Then and While-Do condition statements.* When developing an If-Then sequence or While-Do loop in AVSI II, the construction of Then and Do statement sequences is adequately supported. The If and While conditions, however, have minimal visual support. The condition statements should refer to data values derived from the data in the current subsystem's import or export area. With the current process, the conditions do not have to correspond to export data values, even though they should, and the application specialist's discipline is the only factor preventing nonsensical data from being used as a condition.

5. *Refine the import-export connection process.* Much work remains to be done in the import-export connection process. Several of the user feedback comments mentioned in Section 5.3.3 should be considered for incorporation into AVSI II. The most notable of the suggestions concerns the icon alignment grid. The suggestion to display potential connections between primitives, similar to what is done between subsystems, also warrants consideration. However, the danger exists that the line density on the screen could be intrusive when several primitives with multiple inputs or outputs are involved.

6. *Extend support for the Application Executive.* The Application Executive is a new feature of Architect which was simultaneously being developed with this research (34). Due to the concurrent development of the Application Executive and AVSI II, AVSI II has only rudimentary support for the application specialist to interface with the Application Executive capabilities. Architect's interface needs to be extended to fully support all modes and operations of the Application Executive.

7. *Provide visual support for semantic checks.* As mentioned in the previous chapter, AVSI II does not directly support semantic analysis of an application. A possible feature to incorporate into the interface would be context-sensitive, editable text windows that display the erroneous code and either allow the application specialist to fix the code in the window, or indicate where the error should be fixed by modifications to the icons and links.

8. *Incorporate visualization of application execution.* This item is a significant topic in its own right. The application execution can be visualized in ways as simple

as changing the color or highlight styles of icons as their states change or displaying alternative bitmaps for differing object states. The visualization could be as complex as partial to full animation of the application as it executes.

9. *Investigate Single Window Composition.* The application composition process involves four different windows in which a different aspect of the application definition is conducted. The windows are the Edit-Application-Window, the Edit-Subsystem-Windows, the Edit-Update-Windows, and the Import-Export-Windows (however, event-driven or time-driven modes do not utilize the edit-update-windows). Consideration should be given to adapting AVSI II into a single, or perhaps dual, window composition system. Although such a system would deviate from the OCU-model abstraction, it would be simpler and potentially more intuitive for the application specialist to use. Conceptually, such a process would have the application specialist create and name an application or application-object. Then the specialist would instantiate objects from the technology base and populate a subsystem-composition-window. The objects would then be connected to each other in the manner of AVSI II's import-export connection process of defining the objects' sources and targets. These connections can replace the need to connect objects to a controlling subsystem since the controlling subsystem can be identified by creating a mapping between the composition window and the subsystem-object. The update algorithm can be at least partially inferred from the import-export connection order given that a set of conventions (like left to right data flow) are enforced. The connection order could be used to automatically create a partial dependency graph or table. Additional

inferencing about the application or application specialist inputs would be used to complete the dependency graph.

6.3 Summary

Development of a visual interface is not an exact science. Much of the design effort is spent developing a system that "looks good," "works well," or "seems easy" to use. These concepts are difficult to explain and ever harder to quantify. "Looks good," for example, means different things to different people in different situations. For this type of research, rapid prototyping was the most effective design method available. Incremental development of the interface capabilities helped quantify AVSI II's design by allowing other users to use and comment on the interface as it was evolving.

The role of application composition and generation systems like Architect is increasing in importance within the software industry. One of the keys to the successful development of systems like Architect is an understandable and usable interface. AVSI II extended Architect's usability by incorporating sophisticated graphics and an improved application development processes. As mentioned in this chapter, there is a significant amount of research remaining for the development of an AVSI III or beyond. The interface developer will be a key member of any application composer development team.

Appendix A. Sample Session for AVSI II

This appendix contains a sample session in which a BCD to Excess-3 Decoder is built from the primitive objects defined in the logic-circuits domain. The circuit diagram for the Decoder is shown in Figure A.1.

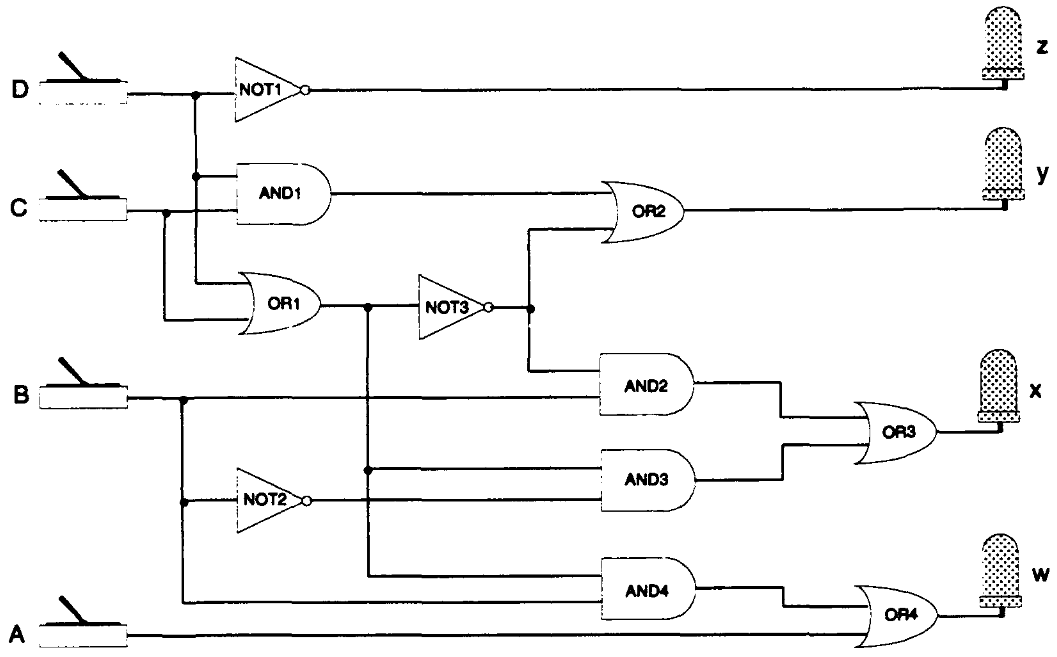


Figure A.1 BCD to Excess-3 Decoder Circuit

A.1 Starting AVSI

REFINE must be loaded in the Emacs window from the Architect directory. Once this is accomplished enter:

```
(load "1")
```

When the prompt returns, enter:

```
(1)
```

It will take several minutes for this file to run because it loads the DIALECT and INTERVISTA systems as well as the Architect and AVSI II files. When the load is complete, a prompt appears:

```
Load Complete

Type "(AVSI)" to start AVSI
```

Now enter the command:

```
(avsi)
```

This action loads the visual specification files for the domains currently defined for Architect. After the visual information is parsed into the object base, the control panel (shown in Figure A.2) appears in the upper left-hand corner of the screen. Across the top of the window is a row of buttons that are used to invoke many of the application composition functions of AVSI II. The lower portion of the window is a message area used to display status and errors.

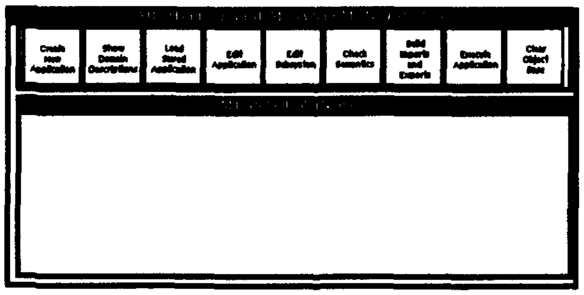


Figure A.2 AVSI II Control Panel

A.2 Create a New Application

To create a new application:

1. Click any mouse button on the button labeled **Create New Application**.
2. A pop-up window appears and prompts, **Select Domain**. Click on the menu item **CIRCUITS**.
3. A pop-up window appears with the prompt, **Enter name of application**. Type

   `bcd-xs3`

4. The name can be entered by hitting the "return" key or by clicking on **Do It** at the bottom of the pop-up window.

A.3 Edit the Application

Now that the application has been created, the next step is to edit the application's elements. Editing an application is comprised of two separate operations: editing an application's components, and editing an application's update algorithm.

A.3.1 To add a controlling subsystem-obj to the application:

1. Click a mouse button on the **Edit Application** control panel button.
2. A pop-up menu appears with the prompt **Choose Application**. Click on the menu item **BCD-XS3**.
3. A pop-up menu appears with the prompt **Choose:** Click on the menu item **Edit Application Components**. A blue window appears containing a single icon labeled, $\frac{\text{APPLICATION - OBJ}}{\text{BCD - XS3}}$ (refer to Figure A.3).

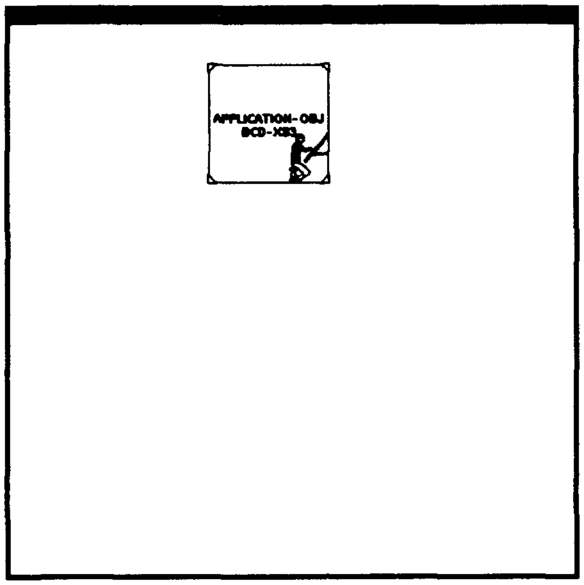


Figure A.3 Edit-Application-Window

4. Click on the diagram surface (anywhere on the blue surface except within the icon's boundary) of the window. A pop-up menu will appear.

5. Select **Create New Subsystem.**

6. A pop-up window appears, with the prompt **Enter a name:**. Enter `driver`

7. A box outline of an icon appears, attached to the mouse cursor. Place the icon below the application-obj icon by moving the cursor to the desired location and clicking.

8. Click any mouse button on the newly created subsystem-obj icon and select the menu option **Link to Source**.

9. The mouse cursor changes from an arrow to an oval with a dot in it, signifying that an object needs to be selected. Place the mouse cursor on the application-obj's icon and click any mouse button. A link appears between the application-obj's icon and the subsystem-obj's icon, as in Figure A.4.

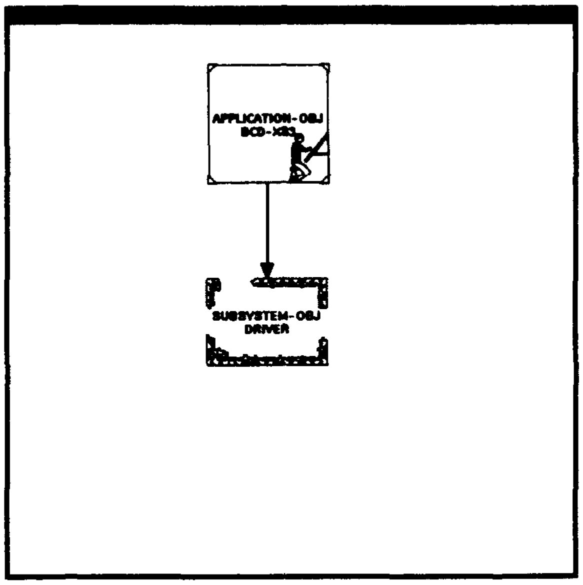


Figure A.4 Edit-Application-Window

10. Close the edit-application-window by clicking on the diagram surface and selecting **Deactivate.**

A.3.2 To create the application-obj's update-algorithm:

1. Click a mouse button on the **Edit Application** control panel button.
2. A pop-up menu appears with the prompt **Choose Application.** Click on the menu item **BCD-XS3.**
3. A pop-up menu appears with the prompt **Choose:**. Click on the menu item **Edit Application Update.** Three windows will appear (refer to Figure A.5). One contains a graphical view of the update algorithm, one contains a textual view of the algorithm, and the third (the Controllee Window) shows the icons that represent the application-obj's controllees (with two extra icons for if-then and while-do con-

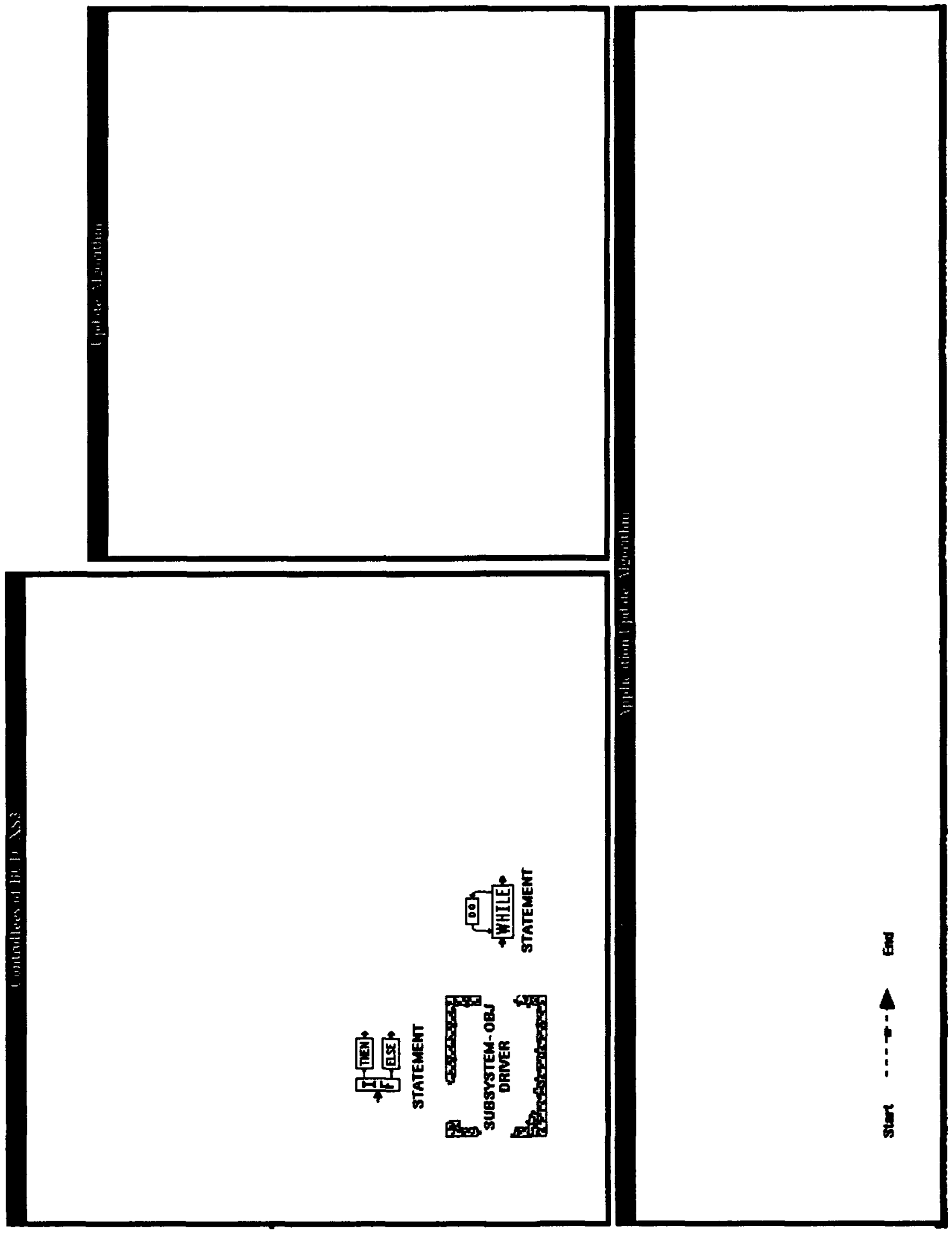


Figure A.5 Edit-Update-Algorithm Windows

structs). The graphical update window contains two icons, "Start" and "End," with a dotted arrow pointing from the start-icon to the end-icon.

4. Click a mouse button on the icon in the controllee window labeled SUBSYSTEM – OBJ DRIVER. The cursor changes to an oval with a dot in it indicating that an object needs to be selected.

5. Click on the "nub" on the dotted line midway between the start and end icons. This will cause the update sequence to redraw with the subsystem-obj included (see Figure A.6). Note the textual representation is automatically updated to reflect each change in the diagram window.

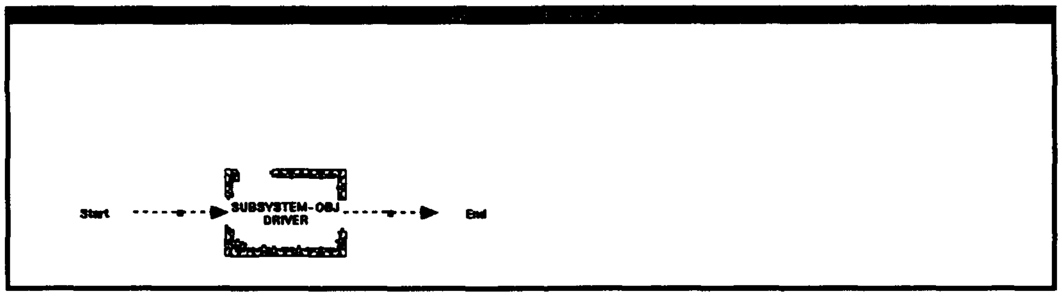


Figure A.6 Edit-Update-Algorithm

6. Close the edit-update-algorithm windows by clicking on the black title bar at the top of the graphical update window and selecting **Deactivate**.

A.4 Edit the Subsystems

Building a subsystem is similar to building the application. This section illustrates instantiating and linking primitive-objs and a nested subsystem-obj to the controlling subsystem created in the previous section.

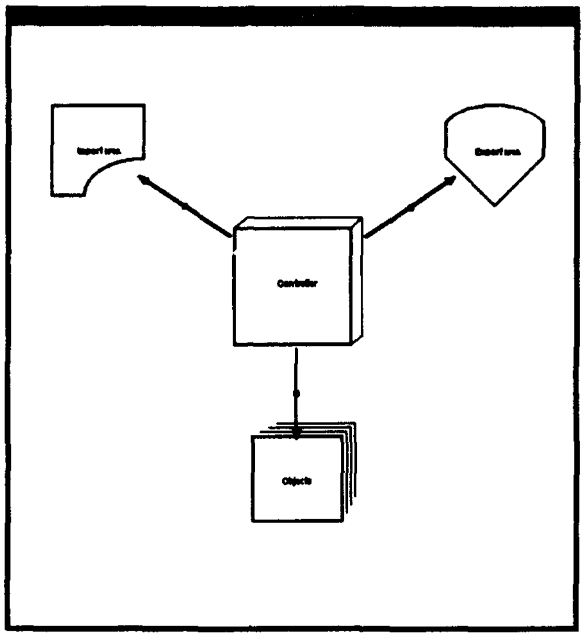


Figure A.7 Subsystem Window

The subsystem, **DRIVER**, will control four switches, four LEDS, and a separate subsystem (called bcd-excess3), which is the heart of the application. Bcd-excess3 consists of three not-gates, four and-gates, and four or-gates (refer back to Figure A.1). To add these objects, perform the following steps:

A.4.1 To add the subsystem:

1. Click on the **Edit Subsystem** button in the control panel window.
2. Click on the menu item **DRIVER**. A white window opens (the subsystem window) which contains an OCU representation of a subsystem (see figure A.7).
3. Click on the **Objects** icon in the subsystem window. The blue edit-subsystem-window for **DRIVER** appears, containing a single icon labeled $\overset{\textbf{SUBSYSTEM - OBJ}}{\textbf{DRIVER}}$.

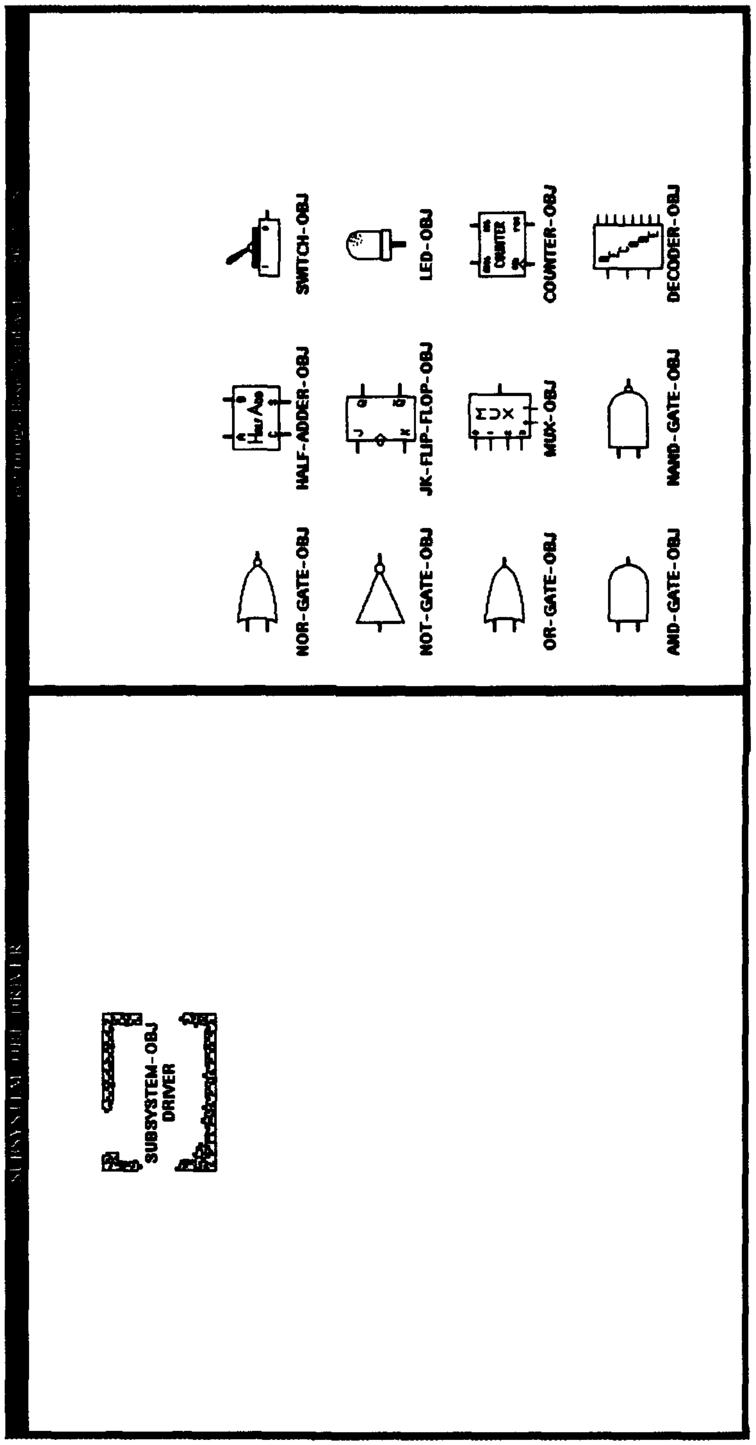


Figure A.8 Edit-Subsystem and Technology-Base Windows

A green window, the technology-base window, also appears and contains an icon for each primitive-object in the current domain (see figure A.8).

4. Click on the blue diagram surface and select the **Create New Subsystem** menu item.
5. Name the new subsystem **bcd-excess3**.
6. Place the subsystem-icon somewhere on the blue window.

A.4.2 To add the primitive objects:

1. Click on the icon in the green technology-base-window labeled **SWITCH-OBJ**, that looks like a toggle switch.
2. A Switch-icon is created and attached to the mouse cursor. Place this icon on in the blue edit-subsystem-window near **DRIVER**.
3. Name the switch by typing **A** in the pop-up window.
4. Follow the above steps to create and place three more switch objects, named **B**, **C**, and **D**.
5. Similarily, create four LED objects named **W**, **X**, **Y**, and **Z**.
6. Link the primitive objects and **BCD-EXCESS3** to **DRIVER** by clicking a mouse button on **DRIVER**, and selecting **Link Multiple Targets** from the pop-up menu.
7. A pop-up window will appear that lists all the unconnected objects in the edit-subsystem-window. Select **All of the Above**, and click on **Do It**. A link will appear from **DRIVER** to each of the other icons. Figure A.9 shows the edit-subsystem-window at this point.

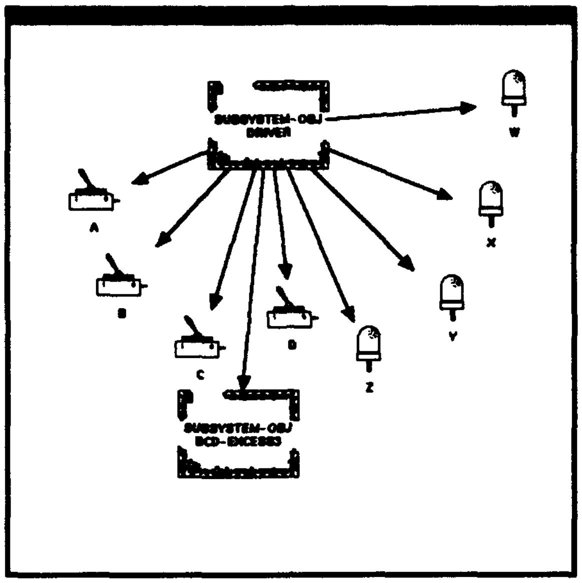


Figure A.9 Edit-Subsystem-Window

8. Now click on the technology-base-window icon that represents a not-gate.

9. Position it near **BCD-EXCESS3** and click a mouse button to "drop" it.

10. Name the not-gate by typing **not1** in the pop-up window.

11. Repeat the process to add **not2** and **not3**.

12. Select the the appropriate icons from the technology-base-window and create **and1**, **and2**, **and3**, **and4**, **or1**, **or2**, **or3**, and **or4**.

13. Link the new primitive objects to **BCD-EXCESS3** by clicking a mouse button on **BCD-EXCESS3**, and selecting **Link Multiple Targets** from the pop-up menu. Select **All of the Above**, and click on **Do It**.

    Note: The window is probably cluttered and disorganized at this point. You can clean up the display by clicking on the blue background of the edit-subsystem-window and selecting **Redraw**. The window is redrawn in an inverted tree-layout with **DRIVER** at the root and the other objects arranged underneath. The entire subsystem will not be visible in the window. You can scroll the window vertically or horizontally using the scroll-bars on the left and bottom of the window. The window can be resized by clicking the mouse on the black title bar and selecting **Reshape**. You can also change the size of the icons in the window by clicking on the blue surface and selecting **Change Scale Factor** from the menu. A new window will appear prompting **Full Size (1:1)** or **Half Size (1:2)**. Select half size and the window will be redrawn with the icons at half scale. You may, at any time, pretty-print an object (show its object base representation) by clicking on its icon and choosing the menu selection **Pretty-Print Object**.

14. Close the edit-subsystem-window and the technology-base-window by clicking on the blue surface and selecting **Deactivate**. The windows can be closed separately by selecting **Deactivate** from their title bar menus.

    At this point, the subsystem window for **DRIVER** will again be visible.

A.4.3 To connect **DRIVER***'s Imports and Exports:* To connect the import and export objects perform the following steps:

1. Click a mouse button on the **Import Area** or **Export Area** icon in **DRIVER**'s subsystem window.
2. Select **Make Connections** from the pop-up window. A red window (the import-export-window) will open and contain the four switch icons, the four led icons, and an OCU-like icon representing the **BCD-EXCESS3** subsystem (as shown in Figure A.10). The black bars (these bars are actually highlighted subicons attached

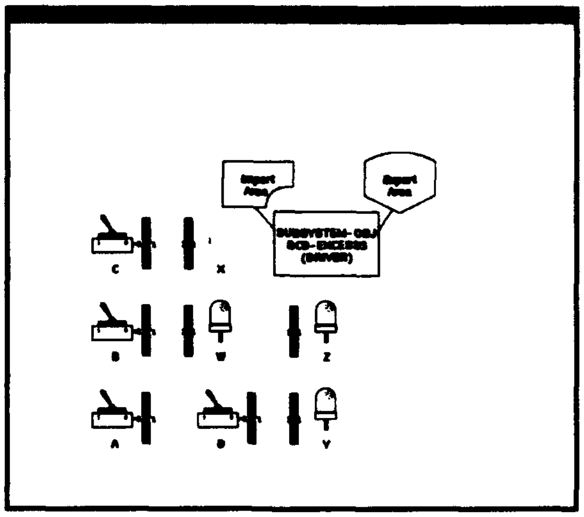


Figure A.10 **DRIVER**'s Import-Export Window

to the primitive's icon) on the sides of the primitive icons indicate connections that need to be made.

3. Icons can be moved to new positions on the screen by clicking on the icon and selecting **Move Icon** from the pop-up menu. An square grid, attached to the mouse, appears. Move the grid to the new icon position and click to "drop" the icon. An example of the repositioned icons is shown in Figure A.11.

4. Click on the part of **BCD-EXCESS3**'s icon labeled **Import Area**. A white text window (a msp-window) will appear in the upper right area of the screen. The window lists the objects in **BCD-EXCESS3** that require input (see Figure A.12).

5. Connect Switch **A** to gate **OR4** by clicking the mouse on the black bar on **A** and then clicking on the text entry **IN2 SIGNAL OR4** in the msp-window. Notice that the black bar changes to a clear box (indicating the switch is connected to an object not visible on the screen), the msp-window is updated to reflect the connection,

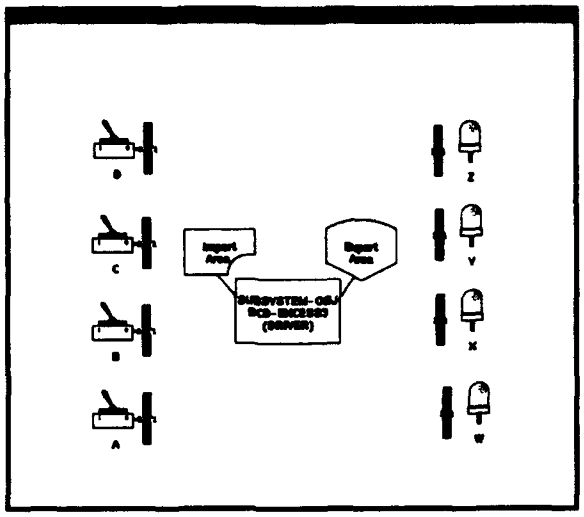
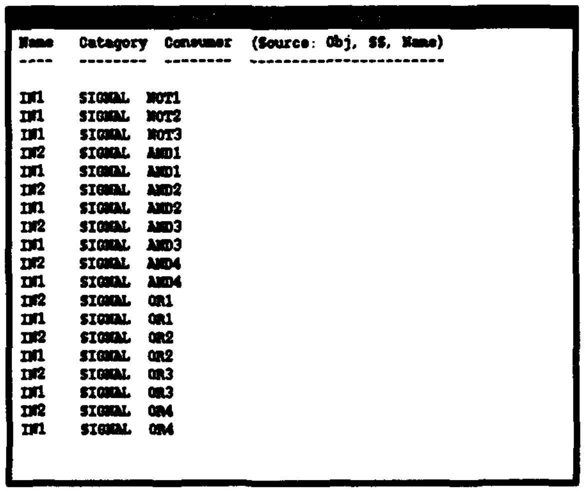


Figure A.11 Repositioned Icons



```
Name   Catagory  Consumer  (Source: Obj, SS, Name)
----   --------  --------  -----------------------


IN1    SIGNAL   NOT1
IN1    SIGNAL   NOT2
IN1    SIGNAL   NOT3
IN2    SIGNAL   AND1
IN1    SIGNAL   AND1
IN2    SIGNAL   AND2
IN1    SIGNAL   AND2
IN2    SIGNAL   AND3
IN1    SIGNAL   AND3
IN2    SIGNAL   AND4
IN1    SIGNAL   AND4
IN2    SIGNAL   OR1
IN1    SIGNAL   OR1
IN2    SIGNAL   OR2
IN1    SIGNAL   OR2
IN2    SIGNAL   OR3
IN1    SIGNAL   OR3
IN2    SIGNAL   OR4
IN1    SIGNAL   OR4
```

Figure A.12 Import Area for **BCD-EXCESS3**

and another msp-window labeled "Export Area for DRIVER" appears. The export window shows the same connections being made from the switch's perspective.

6. In the same fashion, make the following connections:

switch B	connected to	IN1 NOT2
switch B	connected to	IN2 AND2
switch B	connected to	IN2 AND4
switch C	connected to	IN2 AND1
switch C	connected to	IN2 OR1
switch D	connected to	IN1 NOT1
switch D	connected to	IN1 AND1
switch D	connected to	IN1 OR1

   In a similar manner, the leds must be connected.

7. Click on the part of **BCD-EXCESS3**'s icon labeled **Export Area**. A white text window (a msp-window) will appear in the lower right area of the screen. The window lists the objects in **BCD-EXCESS3** that provide output.

8. Connect Led **W** to gate **OR4** by clicking the mouse on the black bar on **W** and then clicking on the text entry **OUT1 SIGNAL OR4** in the msp-window. Again, the black bar changes to a clear box (indicating the led is connected to an object not visible on the screen), the msp-window is updated to reflect the connection, and another msp-window labeled "Import Area for DRIVER" appears. The import window shows the same connections being made from the led's perspective.

9. In the same fashion, make the following connections:

led X	connected to	OUT1 OR3
led Y	connected to	OUT1 OR2
led Z	connected to	OUT1 NOT1

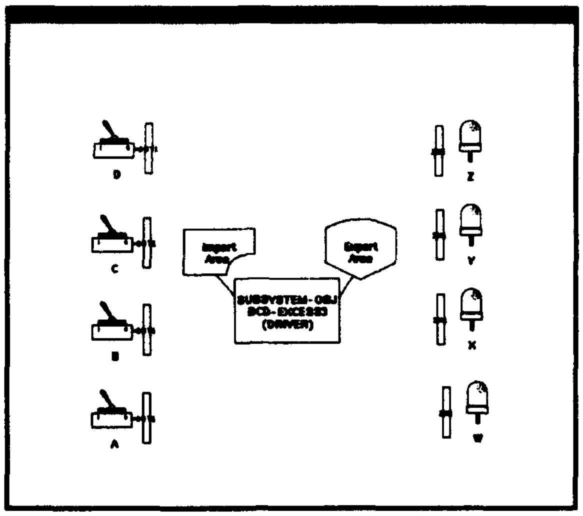


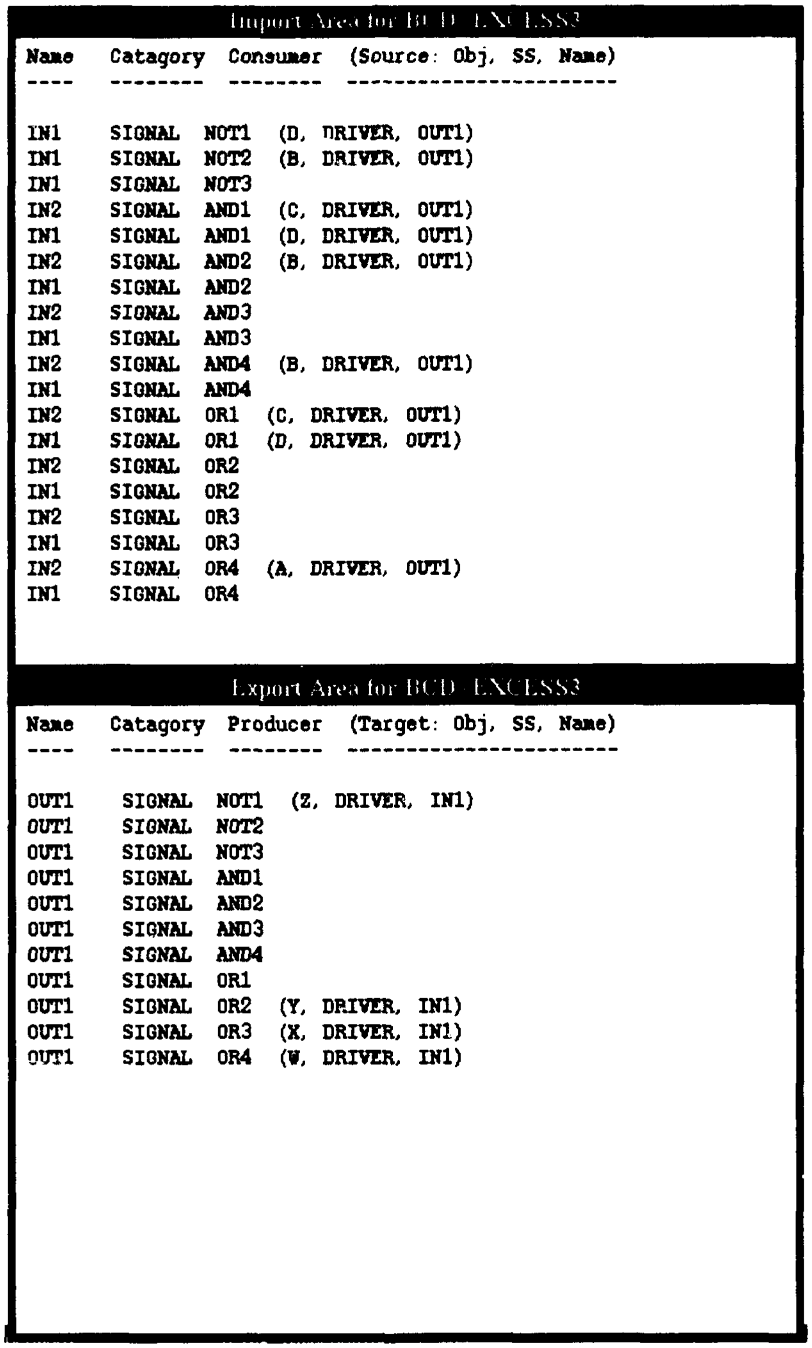
Figure A.13 **DRIVER**'s Import-Export Window

When the connections have been made, the import-export-window should look something like Figure A.13 and **BCD-EXCESS3**'s msp-windows should look like Figure A.14.

10. Close the import-export-window and the msp-windows by clicking on the red surface and selecting **Deactivate**, or by selecting **Deactivate** from each window's title bar menu.

A.4.4 To build **DRIVER**'s *Update Algorithm:* After the import-export windows have been closed, the white subsystem window will again be visible. Building the update algorithm for **DRIVER** is similar to building the update algorithm for the application, and requires the following steps:

1. Click the mouse on the **Controller** icon in the subsystem window. The three windows that were seen in Figure A.5 are exposed, except the controllee window now contains the switches, leds, and subsystem controlled by **DRIVER**.



```
Import Area for BCD-EXCESS3
Name   Catagory  Consumer  (Source: Obj, SS, Name)
----   --------  --------  -----------------------

IN1    SIGNAL  NOT1  (D, DRIVER, OUT1)
IN1    SIGNAL  NOT2  (B, DRIVER, OUT1)
IN1    SIGNAL  NOT3
IN2    SIGNAL  AND1  (C, DRIVER, OUT1)
IN1    SIGNAL  AND1  (D, DRIVER, OUT1)
IN2    SIGNAL  AND2  (B, DRIVER, OUT1)
IN1    SIGNAL  AND2
IN2    SIGNAL  AND3
IN1    SIGNAL  AND3
IN2    SIGNAL  AND4  (B, DRIVER, OUT1)
IN1    SIGNAL  AND4
IN2    SIGNAL  OR1  (C, DRIVER, OUT1)
IN1    SIGNAL  OR1  (D, DRIVER, OUT1)
IN2    SIGNAL  OR2
IN1    SIGNAL  OR2
IN2    SIGNAL  OR3
IN1    SIGNAL  OR3
IN2    SIGNAL  OR4  (A, DRIVER, OUT1)
IN1    SIGNAL  OR4
```

```
Export Area for BCD-EXCESS3
Name   Catagory  Producer  (Target: Obj, SS, Name)
----   --------  --------  -----------------------

OUT1    SIGNAL  NOT1  (Z, DRIVER, IN1)
OUT1    SIGNAL  NOT2
OUT1    SIGNAL  NOT3
OUT1    SIGNAL  AND1
OUT1    SIGNAL  AND2
OUT1    SIGNAL  AND3
OUT1    SIGNAL  AND4
OUT1    SIGNAL  OR1
OUT1    SIGNAL  OR2  (Y, DRIVER, IN1)
OUT1    SIGNAL  OR3  (X, DRIVER, IN1)
OUT1    SIGNAL  OR4  (W, DRIVER, IN1)
```

Figure A.14 Import-Export MSP-Windows

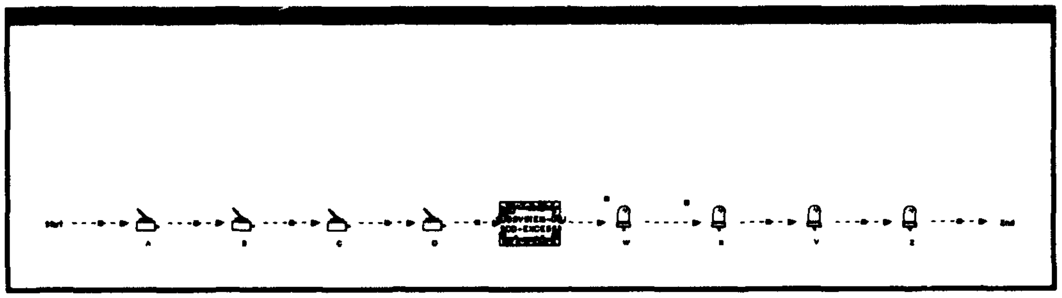


Figure A.15 **DRIVER**'s Update Algorithm

2. Add each controllee to the update sequence by clicking on the controllee icon and then clicking on the "nub" in the graphical update window that represents the proper sequence position for the controllee. The order in which the controllees must appear is:

```
A     B     C     D     BCD-EXCESS3    W    X    Y    Z
```

   When the fifth icon is added to the sequence, the window will automatically resize to keep the entire sequence visible. Figure A.15 shows the completed update algorithm. Note that the textual update description is updated as the graphical update is built.

3. Close the windows by clicking on the graphical update window title bar and selecting **Deactivate.**

A.4.5 To connect **BCD-EXCESS3***'s Imports and Exports:* So far, the application has been created, the controlling subsystem (**DRIVER**) has been created, and **DRIVER**'s components, import-export connections, and update algorithm have been defined. The definition process for **DRIVER** now needs to be repeated for **BCD-**

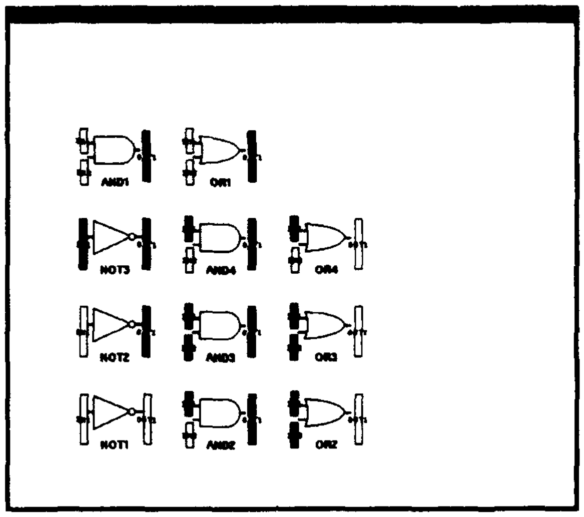


Figure A.16 **BCD-EXCESS3**'s Import-Export Window

EXCESS3. Begin by defining **BCD-EXCESS3**'s import-export connections in the following way:

1. Click a mouse button on the **Edit Subsystem** button on the control panel.
2. Select **BCD-EXCESS3** from the pop-up window. A subsystem window will open. This window correspondes to **BCD-EXCESS3**, as indicated by the window title. Although the **BCD-EXCESS3**'s components were defined in Section A.4.2, they can be viewed by clicking on the subsystem's **Objects** icon. The windows are closed by clicking on the blue surface and selecting **Deactivate**.
3. Open the import-export-window by clicking the mouse on the **Import Area** or **Export Area** icon and selecting **Make Connections** from the pop-up window. The import-export-window will look like Figure A.16. As with the earlier import-export-window, the black bars (again, actually subicons) represent connections to be

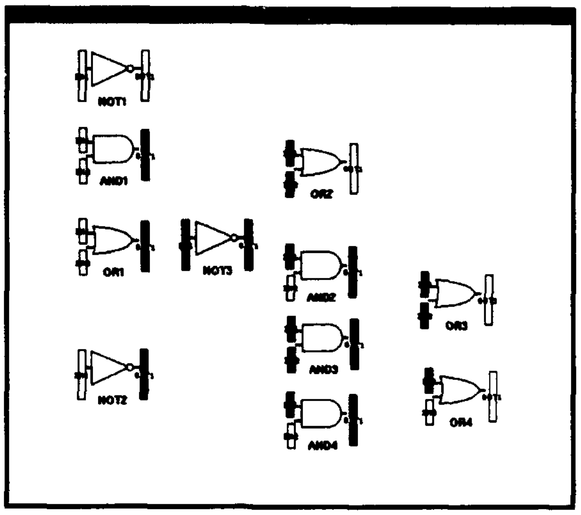


Figure A.17 **BCD-EXCESS3**'s Import-Export Window

made, and the clear boxes (also subicons) represent connections made to objects not visible in the window.

4. Reposition the icons to resemble the circuit diagram in Figure A.1. The icons are repositioned by clicking on the icon, selecting **Move Icon** from the pop-up menu, moving the grid to the desired location, and clicking to "drop" the icon. The repositioned icons will probably look something like Figure A.17.

5. Connect **AND1** to **OR2** by clicking on the black bar labeled OUT1 on **AND1** and then clicking on the black bar labeled IN1 on **OR2**. Note that a line (a link) appears between the icons and the black bars disappear. The subicons of the primitives are still present, and are highlighted by a thin white line when the mouse is over one.

6. Make the remaining connections described below:

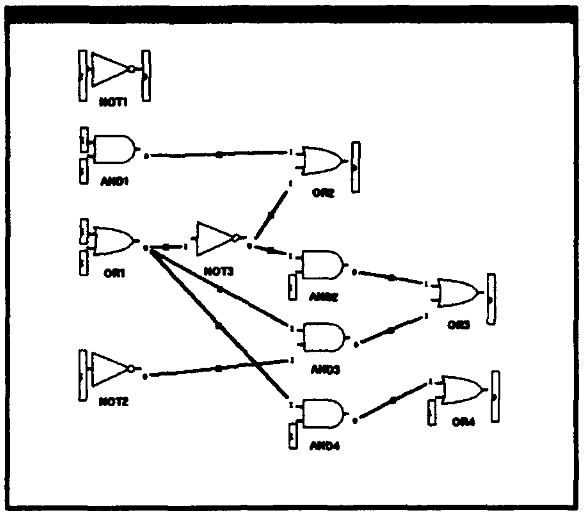


Figure A.18 **BCD-EXCESS3**'s Import-Export Window

AND1 OUT1	connected to	OR2 IN1
OR1 OUT1	connected to	NOT3 IN1
OR1 OUT1	connected to	AND3 IN1
OR1 OUT1	connected to	AND4 IN1
NOT2 OUT1	connected to	AND3 IN2
NOT3 OUT1	connected to	OR2 IN2
NOT3 OUT1	connected to	AND2 IN1
AND2 OUT1	connected to	OR3 IN1
AND3 OUT1	connected to	OR3 IN2
AND4 OUT1	connected to	OR4 IN1

Once the connections have been made, the import-export window should resemble Figure A.18. If desired, the links, once made, can be "hand-drawn" by clicking on the link's nub and selecting **Re-Draw Path.** The existing link will be deleted, the mouse cursor will change to a "pencil," and a dashed line will connect the icon to the cursor. Draw the link by moving the cursor to a new position and clicking the mouse button. To finish drawing the link, click the mouse button on a subicon. Also, if you find the labels on the subicons intrusive, they can be suppressed by clicking on the window surface and selecting **Clip Icon Labels** from the pop-up menu.

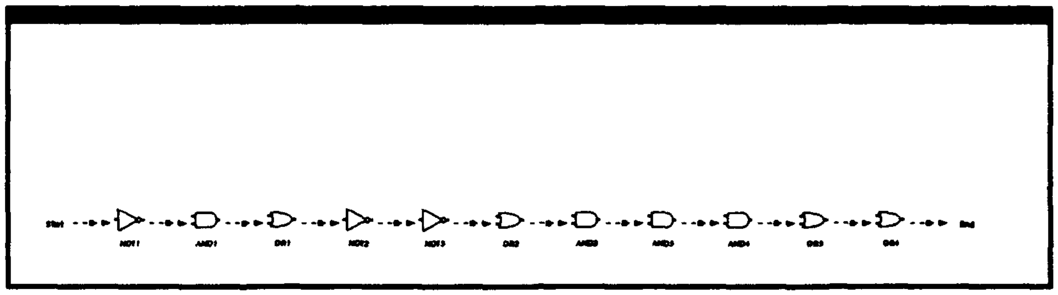


Figure A.19 **BCD-EXCESS3**'s Update Algorithm

7. Close the import-export-window by clicking on the red surface and selecting **Deactivate** from the menu.

A.4.6 To build **BCD-EXCESS3** *'s Update Algorithm:* When the import-export-window closes, the subsystem window will be visible. To build the update algorithm for **BCD-EXCESS3** do the following:

1. Click the mouse on the **Controller** icon i. the subsystem window. The three update windows will be exposed, and the controllee window will contain the controllees defined for **BCD-EXCESS3**.

2. Create the update sequence by clicking the mouse on an icon in the controllee window and then clicking on the nub in the graphical update window as was done with the application and **DRIVER** update algorithms. Add the controllees to the update algorithm in the following sequence:

```
NOT1  AND1  OR1  NOT2  NOT3  OR2  AND2  AND3  AND4  OR3  OR4
```

The completed sequence is shown in Figure A.19.

Table A.1 BCD to Excess-3 Decoder Truth Table

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

3. Close the update windows by selecting **Deactivate** from the title bar menu of the graphical update window.

A.5 Perform Semantic Checks

Semantic checks are performed by Architect as part of the import-export connection process. However, the semantic checks may be run at any time by clicking on the control panel button labeled **Check Semantics**. The results of the semantic checks may be viewed in the EMACS window.

A.6 Execute the Application

Now that the application has been fully defined, it can be executed. The truth table for the BCD-XS3 decoder is shown in Table A.1.

The default position for a switch is "on," however 1 1 1 1 is not a defined input for a BCD to Excess3 Decoder. Therefore, the switch attributes have to be set to a meaningful input in order to get a valid output from the execution. To set the switch attributes, do the following:

1. Click on the control panel's **Edit Subsystem** button.
2. Choose **DRIVER** from the pop-up window.
3. Click on the **Objects** icon in the subsystem window.
4. Click on the switch icon labeled **A**.
5. Choose **View/Edit attributes** from the pop-up window. A window appears, listing the switch attributes for **A**.
6. Click on the attribute, **Position**. A pop-up window appears, listing the current value of the switch.
7. Enter a new value for the switch position by typing

   `avsi::off`

   (The "avsi::" prefix is the package name and is required for symbols. It is not required for other data types such as numbers and strings)
8. Change the values for any other switches in the same manner as above.

Changing the switch settings can also be accomplished in the edit-application-window by clicking on the **Edit Application** control panel button, selecting **BCD-XS3** from the list, selecting **Edit Application Components** from the menu, and then making the attribute changes as described above. Opening the edit-application-window will alse allow you to see the entire application in tree-layout format. Window reshaping and icon resizing will be necessary to view the entire application. Figure A.20 shows the application.

Once a valid input state has been achieved, click on the control panel button labeled **Execute Application**. The results are displayed in the EMACS window. A new set of switch positions can be established, and the application re-executed.

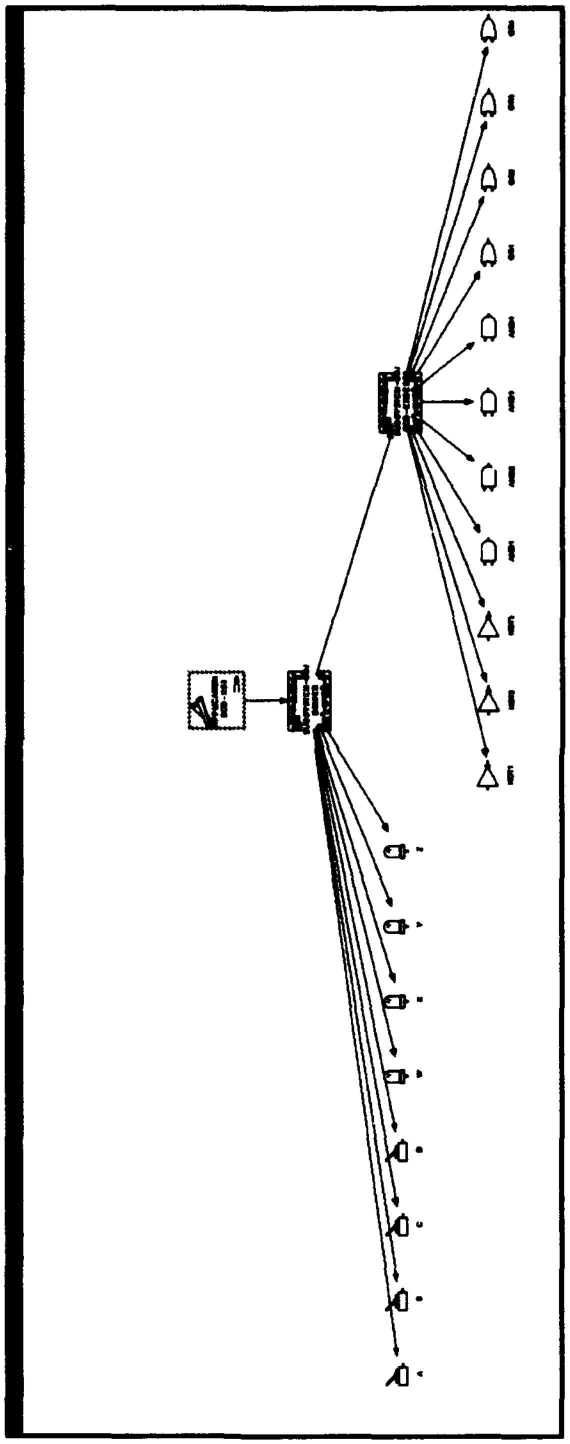


Figure A.20 **BCD-XS3** Application

Appendix B. Instructions for Developing a Metaphor Set

This appendix contains information that a domain modeller or domain engineer will find useful when developing a new domain for Architect. The following sections discuss some design considerations to be used in determining icon metaphors for domain primitive objects, as well as detailed instructions about the naming conventions and file requirements for the Architect Visual System Interface (AVSI) II.

B.1 Design Principles

Webster's New Collegiate Dictionary defines an icon as "a usually pictorial representation: IMAGE" (19). Although icons are frequently associated with religious imagery, an icon is, in fact, any picture or symbol used to represent another object, idea, or action. Icons have infiltrated almost every aspect of our daily lives, especially as a result of attempts to communicate across the barriers of language. Icons should convey precise messages simply through their shape or picture; any accompanying text is usually unnecessary. This capability to convey meaning is reflective of Arend's concept of a "global superiority" visual effect (2). The global superiority effect asserts that global features like shape, size, and color can be recognized and assimilated faster than features like lines and text. In other words, simple pictures with one or two distinctive features make effective icons. Nan Shu also endorses the use of icons by stating (25):

- Pictures are more powerful than words as a means of communication because they convey more meaning in a more concise unit of expression.

- Pictures aid comprehension and remembering.

- Pictures typically do not have language barriers. Properly designed pictures are understood by people regardless of their native language.

The primary goal of a Graphical User Interface (GUI) is to increase human productivity and speed, and to reduce human error and fatigue. In *Human-Computer Interface Design Guidelines*, Brown notes that appropriately applied icons and symbols, directly manipulatable by a user in ways analogous to reality, can help the user maintain context and orientation and reduce the user's need to memorize commands and syntax. More importantly, looking at and working with pictures is more fun than working with a screen full of text. Icons can be more effective than text because they can convey a large amount of information in little space, they are easily learned and easily remembered, and they are more universal (3). The icons and symbols on a display are frequently referred to as metaphors because of the designed intention of using them to represent a piece of reality in an abstract way. The icon or metaphor is a visual representation of a command, task, method, file, object, etc. The first challenge in GUI design is, therefore, to develop a set of metaphors appropriate to the domain of interest.

Determining what metaphor to use as an icon can be difficult. The key to developing a useful metaphor is to exploit a user's knowledge of familiar objects in a domain. Erickson recommends looking at a system's functionality for clues. He says to "note what metaphors are already implicit in the problem description [domain]" because people commonly use metaphors when talking about abstract concepts, and "it's almost certain that metaphors are lurking about in the description of the [system's] functionality" (9:69).

Many authors have attempted to describe what constitutes a good icon design. Deborah Mayhew recommends the following (18:316-330):

- Minimize semantic and articulatory distance,
- Provide visual feedback for icon position, selection, and movement,
- Choose a consistent icon scheme,
- Design icons to be concrete and familiar,
- Design icons in a set to be visually and conceptually distinct,
- Avoid excessive detail in icon design,
- Accompany icons with names,
- Limit the number of icon types to 12 if possible, but at most 20.

Semantic distance is the subjective or psychological distance between the user's intended action on an icon and the operations allowed by the interface to be performed on the icon. Semantic distance directly applies to the level of abstraction represented by the GUI. Articulatory distance refers to the relationship between an icon's shape and its meaning. For example, representing a Nand-gate with an icon that looks like a baseball imposes a large articulatory distance on the metaphor.

Ben Shneiderman recommends many of the same guidelines, but adds that the designer should (24:210):

- Make the icon stand out from its background,
- Ensure the harmoniousness of each icon as a member of a family of icons.

Mayhew's fifth guideline (i.e., design icons in a set to be distinct) and Shneiderman's second guideline (i.e., ensure the harmony of each icon in a family) relate well. Within a specific domain (or set or family) every icon must be distinct and recognizable from every other icon, but they must all relate to the overall domain. A paintbrush would be out of place in a logic-circuits domain, for example.

Carroll, Mack, and Kellogg state that developing a set of interface metaphors, or icons, is an iterative process that is refined over time. They claim the first step in creating a metaphor set is to generate a set of candidate metaphors and develop them using these ingredients:

- User's experiences with other systems or situations similar to the system being designed,
- User's knowledge, perceptions, and motivations,
- Sheer invention.

They further emphasize that they know of no formal analysis method for generating useful metaphors (6).

Chang goes a step further by recommending a strategy for creating visual representations of objects. He recommends considering the following factors (7:69-70):

- Application and Cultural Dependence: Icons typically represent a class of objects rather than one specific instance. Avoid excessive detail and reflect the semantic nature of an object.

- Easy Recognition: An icon should have a precise meaning that is easily recognized once identified. Icons do not have to be read, and people typically do not read words they are familiar with anyway (Note: This does not preclude icon labels). There are more distinctively shaped pictures than words.
- Distinction From Other Icons Within the System: An icon must be individually distinctive and should use a distinguishing feature to reflect the meaning it represents.
- Consistency in an Environment: Although icons should be distinctive, they should be consistent and harmonious within a system or domain.

Erickson describes a process for generating interface metaphors which consists of the following fundamental steps (9:68-72):

- Determine Functional Definition: It is necessary to understand something in order to develop a model/metaphor for it. This includes what it looks like, how it functions, and what its semantic behavior is.
- Identify User's Problems: Figure out what information about an object is important to a user and incorporate the information into the icon.
- Generate Metaphors: Note what metaphors are already implicit in the problem (or domain) description. Using previously identified metaphors can aid recognition of the icon.
- Evaluate Interface Metaphors: Once the metaphors have been developed, evaluate their accuracy and effectiveness through actual use. Allow users to provide feedback on the icon design.

Developing the proper metaphors to represent the primitive objects of a specific domain requires the domain modeller or engineer to have extensive knowledge of the domain and of the people who will be interacting with the domain.

B.2 Bitmap Construction

B.2.1 Available Tools. Once the metaphors have been developed, the next step is to create bitmaps to "visualize" the metaphors. The actual icons (bitmaps) for a domain's primitives can be developed using Icon Editor, an OpenWindows^{TM} tool, or any paint program capable of generating an X11 Bitmap format file. Figure B.1 shows the Icon Editor tool, and the drawing tools associated with it. The program has a small window that shows the icon at it's actual size, and a large window where a pixel by pixel edit of the icon can be accomplished. Make sure the "B&W" button is selected and that the format is "X bitmap." The format is set from the "Format" submenu under the "Properties" menu.

B.2.2 Required Sizes. AVSI II currently supports two display sizes for the objects in a window. Full-sized (the default) icons are displayed using a 64x64 pixel bitmap. Half-sized icons are displayed using a 32x32 pixel bitmap. The "Size" submenu under the "Properties" menu is were the size is set. Each primitive must have a large, or full-sized, and a small, or half-sized, bitmap file created. Using sizes other than 64x64 and 32x32 is not recommended because the AVSI II window refresh functions expect bitmaps of those sizes and will not draw other sized bitmaps in the proper location. The next section discusses the naming conventions used for the bitmap files and how to declare which files contain what bitmaps to AVSI II.

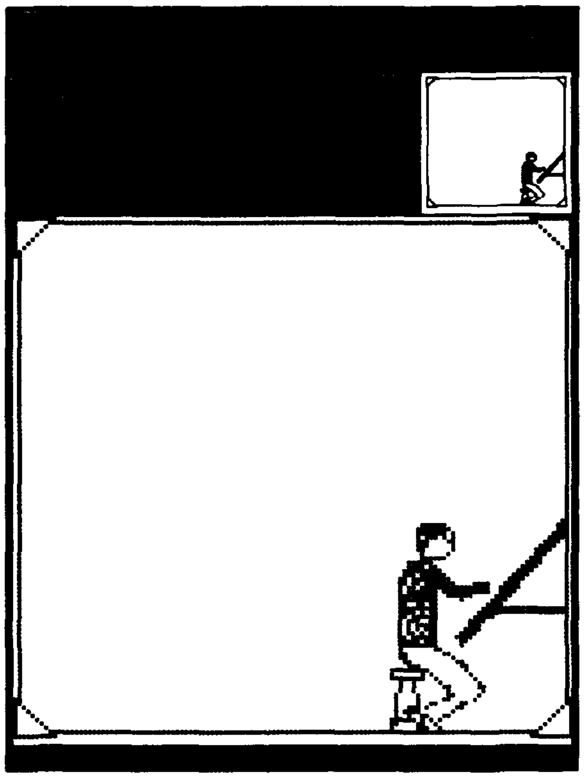


Figure B.1 The Icon Edit Utility

B.3 File Creation

B.3.1 <bitmap>*-l.* This is the recommended naming style for the full-sized (64x64 pixel) bitmap. If the file has been properly created, the file header will look similar to this and-gate example:

```
#define andgate.icon_width 64
#define andgate.icon_height 64
static char andgate.icon_bits[] = {
```

B.3.2 <bitmap>*-s.* This is the recommended naming style for the half-sized (32x32 pixel) bitmap. If the file has been properly created, the file header will look similar to this and-gate example:

```
#define andgate.icon-s_width 32
#define andgate.icon-s_height 32
static char andgate.icon-s_bits[] = {
```

B.3.3 var-<domain>. This file contains the function call that reads a bitmap file from the system and assigns a variable name to represent the bitmap-stream (the actual internal graphic) that is created. The relative path and complete filename must appear in the quotation marks. The UNIX file system is case sensitive, so the filename must be exact. The declarations for the and-gate primitive are given as an example:

```
var and-bitmap-l    : any-type = (cw::read-bitmap(
                                  "./CIRCUITS-TECH-BASE/andgate.icon-l"))
var and-bitmap-s    : any-type = (cw::read-bitmap(
                                  "./CIRCUITS-TECH-BASE/andgate.icon-s"))
```

Note that there are two declarations for the and-gate; one for the large and one for the small bitmaps.

B.3.4 vsl-<domain>. This file declares the user editable attributes and icon attributes for each primitive object in the domain. The relevant entries in this file are the **bitmap4icon-l** and **bitmap4icon-s** lines. The entries on the right side of the "=" must exactly match the variable names declared in the var-<*domain*> file discussed above. The vsl-<*domain*> entries for the and-gate are shown below.

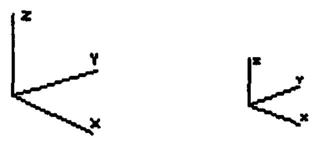
```
attributes for And-Gate-Obj are
Icon :
     label = class-and-name;
     clip-icon-label? = false;
     border-thickness = 0;
     bitmap4icon-l = and-bitmap-l;
     bitmap4icon-s = and-bitmap-s
Edit :
     name : symbol;
     delay : integer;
     manufacturer : string;
     mil-spec? : boolean;
     power-level : real
end;
```

B.4 Conclusion

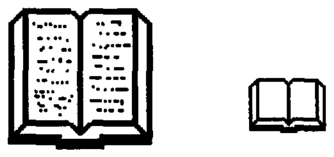
The task of drawing the bitmaps and integrating them into AVSI II is not complicated. The real challenge to the domain modeller or engineer is to develop a set of useful and meaningful metaphors to represent the primitive objects.

Appendix C. A General Collection of AVSI II Metaphors

This appendix presents a set of general purpose icons. Each bitmap pair is saved in files with the name of the icon and a -l or -s suffix for large (64x64 pixel) or small (32x32 pixel) size, respectively. Each pair is followed by a short description of the intended or possible metaphors inherent in the shape. The bitmaps were constructed using the process described in Appendix B. All bitmaps are available in the "generic-icons" subdirectory of the Architect directory.

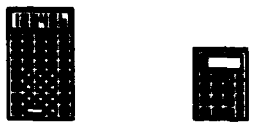


axis.icon : This bitmap is a three dimensional coordinate axis. The metaphor could represent a physical plot, a spatial relationship, or a volume.

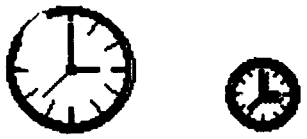


book.icon : This bitmap is an open book. The possible metaphors represented are reference material, information or help, a source of data, or a collection of processed data.

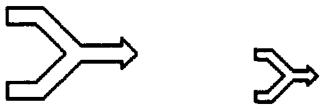
bulb.icon : This bitmap is a light bulb. Traditionally a metaphor for an idea, this icon could also represent a light source or a new concept.



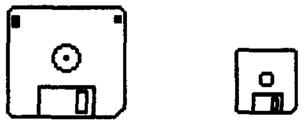
calculator.icon : This bitmap is a calculator. The metaphor can represent any mathematical, financial, or statistical manipulation.



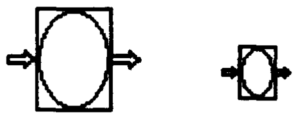
clock.icon : This bitmap is a clock. The metaphor can represent the passage of time, a deadline, or an appointment/meeting/rendezvous.



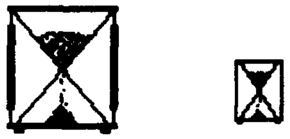
combine.icon : This icon's metaphor is the combination of two things. Possibilities for using the icon include combining data, fluid flows, air flows, or other objects.



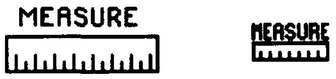
disk.icon : This icon is a disk. Its metaphor represents storage of data, transportation of data, or an external source of data.



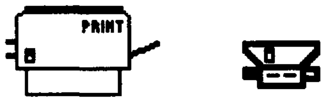
process.icon : This bitmap's metaphor can represent any process (generic, specific, abstract, or concrete) where something is taken in, possibly manipulated , and output.



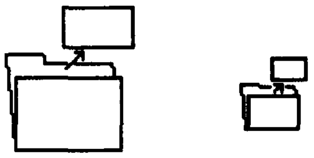
hourglass.icon : This icon is an hourglass. Similar to the clock, this metaphor can represent the passage of time, but on a smaller scale. It can also represent a visible passage of time – meaning slowness.



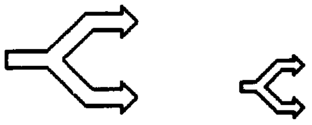
measure.icon : This bitmap is a ruler. It can represent length, distance, size, or some other physical measurement.



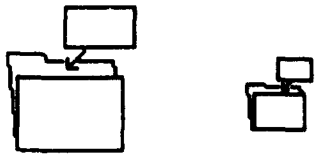
printer.icon : This bitmap is a printer. The metaphoric representation is data output.



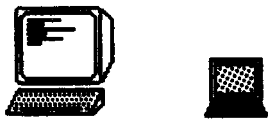
retrieve.icon : This bitmap is a piece of paper being removed from a folder. It represents data or information retrieval. It also implies the retrieval is of small amounts of data from a nearby or local source.



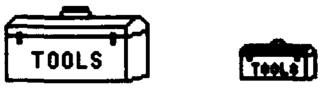
split.icon : This bitmap is a companion to **combine.icon**. It represents the division of some item into parts.



store.icon : This bitmap is a companion to **retrieve**. The metaphor represents the saving or storage of information.



terminal.icon : This bitmap is a computer terminal or monitor. The metaphor represents the display of information or data; or a visual output source.



toolkit.icon : This bitmap is a tool-box. It can represent a variety of programming, design, or construction aids, *utilities*, *or tools*.

In addition to the bitmaps shown above, there is a wide variety of two- and three-dimensional geometric shapes that can be used as metaphors. Examples include rectangles, boxes, circles, diamonds, triangles, cubes, and spheres.

Appendix D. REFINE *Code Listings for AVSI II*

This appendix contains a listing of the Lisp files required to run Architect and AVSI II. The order in which the files are iisted below also indicates the required compilation order.

```
% Load system files for Dialect and Intervista

  (load-system "dialect" "1-0")
  (load-system "intervista" "1-0")
  (load "AVSI-pkg.fasl4")

% Load Architect files

  (in-package 'AVSI)

  (load "DSL/lisp-utilities.fasl4")
  (load "OCU-dm/dm-ocu.fasl4")
  (load "OCU-dm/gram-ocu.fasl4")
  (load "DSL/globals.fasl4")
  (load "DSL/obj-utilities.fasl4")
  (load "DSL/read-utilities.fasl4")
  (load "DSL/menu.fasl4")
  (load "DSL/display-files.fasl4")
  (load "DSL/modify-obj.fasl4")
  (load "DSL/save.fasl4")
  (load "DSL/generic.fasl4")
  (load "DSL/build-generic.fasl4")
  (load "DSL/complete.fasl4")
  (load "DSL/set-globals.fasl4")
  (load "OCU/imports-exports.fasl4")
  (load "OCU/descriptor-tools.fasl4")
  (load "OCU/eval-expr.fasl4")
  (load "OCU/execute.fasl4")
  (load "OCU/semantic-checks.fasl4")
  (load "lisp-file-utils.fasl4")

% Load AVSI II files

  (load "vsl-dm.fasl4")
  (load "vsl-gr.fasl4")
  (load "vsl-globals.fasl4")
  (load "vsl-utils.fasl4")
  (load "viz-utils.fasl4")
  (load "edit-expression.fasl4")
  (load "edit-update.fasl4")
  (load "edit-attr.fasl4")
  (load "create-obj.fasl4")
  (load "edit-ss.fasl4")
  (load "edit-applic.fasl4")
  (load "tech-base.fasl4")
  (load "imp-exp.fasl4")
  (load "test-primitive.fasl4")
  (load "app-exe.fasl4")
  (load "viz.fasl4")
  (load "AVSI.fasl4")

% Load Application Domain-Specific Files Here

  )
```

The REFINE source code for AVSI II may be obtained, upon request, from:

Maj Paul Bailor
AFIT/ENG
2950 P Street
Wright-Patterson AFB, OH 45433-7765

(513)255-9263
DSN 785-9263
email: pbailor@afit.af.mil

Bibliography

1. Anderson, Cynthia. *Creating and Manipulating Formalized Software Architectures to Support a Domain-Oriented Application Composition System.* MS thesis, AFIT/GCS/ENG/92D-01, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.
2. Arend, U. and others. "Evidence for global feature superority in menu selection by icons," *Behavior and Information Technology*, *6*:411–426 (January 1987).
3. Baecker, Ronald M. and William A.S. Buxton. "Chapter 10 - Introduction." *Readings in Human-Computer Interaction: A Multidisciplinary Approach* edited by Ronald M. Baecker and William A.S. Buxton, 427–437, Morgan Kaufmann Publishers, Inc., 1987.
4. Baecker, Ronald M. and William A.S. Buxton. "Chapter 7 - Introduction." *Readings in Human-Computer Interaction: A Multidisciplinary Approach* edited by Ronald M. Baecker and William A.S. Buxton, 299–307, Morgan Kaufmann Publishers, Inc., 1987.
5. Brown, C. Martin. *Human-Computer Interface Design Guidelines.* Norwood, New Jersey: Ablex Publishing Corp., 1988.
6. Carroll, John M. and others. "Interface Metaphors and User Interface Design." *Handbook of Human-Computer Interaction* edited by Martin Helander, 67–85, North-Holland, 1988.
7. Chang, Shi-Kuo. "Principles of Visual Languages." *Visual Programming Systems* edited by Shi-Kuo Chang, 1–59, Prentice Hall, 1990.
8. Dinan, John A. "Human-Interface Rules Reduce Test-Program Operator Errors," *EDN*, *37*:95–98 (February 3 1992).
9. Erickson, Thomas D. "Working with Interface Metaphors." *The Art of Human-Computer Interface Design* edited by Brenda Laurel, 65–73, Addison-Wesley Publishing Co., 1991.
10. Franz Incorporated. *Allegro Common Windows on X Manual.* Berkeley, CA, 1990.
11. Gool, Warren E. *Alternative Architectures for Domain-Oriented Application Composition and Generation Systems.* MS thesis, AFIT/GCS/ENG/93D-11, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.
12. Hewlett-Packard Laboratories. *Release Notes for WINTERP.* Palo Alto, CA, 1990.
13. Inc., Reasoning Systems. INTERVISTA^{TM} *User's Guide.* Palo Alto, CA, 1991. For INTERVISTA^{TM} Version 1.0.
14. Kantrowitz, Mark and Barry Margolin. "FAQ: Lisp Window Systems and GUIs 7/7 [Monthly Posting]," *comp.lang.lisp Newsgroup on NNTP Newsnet* (August 1993).
15. Lee, Kenneth J. and others. *Model-Based Software Development (Draft).* Technical Report CMU/SEI-92-SR-00, Software Engineering Institute, December 1991.
16. Lowry, Michael R. "Software Engineering in the Twenty-first Century," *AAAI*, *13*:71–87 (Fall 1992).

17. Mano, M. Morris. *Digital Logic and Computer Design*. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1979.

18. Mayhew, Debora J. *Principles and Guidelines in Software User Interface Design*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1992.

19. Merriam-Webster, A. *Webster's New Collegiate Dictionary*. Springfield, Massachusettes: G. and C. Merriam Co., 1981.

20. Mike Spenke. *GINA User Manual*. German National Research Center for Computer Science, Germany, 1992.

21. Murch, Gerald M. "Colour Graphics - Blessing or Ballyhoo?:" *Readings in Human-Computer Interaction: A Multidisciplinary Approach* edited by Ronald M. Baecker and William A.S. Buxton, 333–341, Morgan Kaufmann Publishers, Inc., 1987.

22. Randour, Mary Anne. *Creating and Manipulating a Domain-Specific Formal Object Base to Support a Domain-Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/92D-13, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1992.

23. Salomon, Gitta. "New Uses for Color." *The Art of Human-Computer Interface Design* edited by Brenda Laurel, 269–278, Addison-Wesley, 1991.

24. Shneiderman, Ben. *Designing the User Interface: Strategies for Effective Human-Computer Interaction, Second Edition*. New York: Addison-Wesley Publishing Co., 1992.

25. Shu, Nan. *Visual Programming*. New York: Van Nostrand Reinhold Company, 1988.

26. Symbolics Inc. *Common Lisp Interface Manager Release 2.0 Specification*. Burlington, MA, 1992.

27. Texas Instruments, Inc. *CLX: Common LISP X Interace*. Dallas, TX, 1989.

28. Texas Instruments, Inc. *Release Notes for CLUE*. Dallas, TX, 1990.

29. Tullis, Thomas S. "Screen Design." *The Art of Human-Computer Interface Design* edited by Brenda Laurel, 377–411, Addison-Wesley, 1991.

30. Verplank, William L. "Graphic Challenges in Designing Object-oriented User Interfaces." *Handbook of Human-Computer Interaction* edited by Martin Helander, 365–376, North-Holland, 1988.

31. Waggoner, Robert W. *Domain Modeling of Time-Dependent Systems*. MS thesis, AFIT/GCS/ENG/93D-23, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

32. Warner, Russel M. *A Method for Populating the Knowledge Base of AFIT's Domain Oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/93D-24, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

33. Weide, Timothy. *Development of a Visual System Interface to Support a Domain-oriented Application Composition System*. MS thesis, AFIT/GCS/ENG/93M-05,

School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, March 1993.

34. Welgan, Robert L. *Domain Analysis and Modeling of a Model-Based Software Executive*. MS thesis, AFIT/GCS/ENG/93D-25, School of Engineering, Air Force Institute of Technology(AU), Wright-Patterson AFB, OH, December 1993.

35. Williams, Tom. "Graphics Interfaces Make Knobs and Switches Obsolete," *Computer Design*, *29*:78–94 (August 1 1990).

Vita

Captain Jay A. Cossentine was born December 31, 1961 in Ada, Minnesota and graduated from Maryville High School in Maryville, Tennessee in 1980. He was awarded a Bachelor's Degree in Electrical Engineering and commissioned in the United States Air Force through the Air Force Reserve Officer Training Program at the University of Tennessee, Knoxville, in December 1984 . He spent February 1985 through September 1989 as a program manager and project engineer at the Air Force Plant Representative Office (AFPRO), Boeing Military Airplane Company, Wichita KS, on various programs including the Air Force One Replacement Program. From September 1989 to January 1990, he supported the F-117 and several other System Program Offices at the Aeronautical Systems Division, Wright-Patterson AFB, OH. In January 1990 he was assigned as lead avionics engineer for the F-16 Night Attack/Close Air Support program. In March, 1992, he entered the Air Force Institute of Technology at Wright-Patterson AFB, Ohio to pursue a Master of Science degree in Computer Science.

Permanent address: 5610 Morganton Rd
Greenback, Tn 37742

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1993	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE DEVELOPING A SOPHISTICATED USER INTERFACE TO SUPPORT DOMAIN-ORIENTED APPLICATION COMPOSITION AND GENERATION SYSTEMS	5. FUNDING NUMBERS
6. AUTHOR(S) Jay A. Cossentine	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/93D-04
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Capt Rick Painter 2241 Avionics Circle, Suite 16 WL/AAWA-1 BLD 620 Wright-Patterson AFB, OH 45433-7765	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Distribution Unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research refined the visual presentation and usability of a previously developed visual interface for a domain-oriented application composition and generation system. The refined visual interface incorporated domain-specific bit-mapped graphics and sophisticated user interface design concepts to reduce user workload. User workload was reduced through object layout, window design, and color utilization techniques; by combining repetitive procedures into single commands; and reusing, rather than recreating, composition information throughout the application composition process. The Software Refinery environment, including its graphical interface tool INTERVISTA, was used to develop techniques for visualizing and manipulating objects contained in a formal object base. INTERVISTA was supplemented with graphical routines provided by Common Windows, a Lisp-based graphical environment that serves as the foundation for INTERVISTA. The interface was formally validated with a well-understood application domain, digital logic-circuits, and users of the interface were polled to ascertain the subjective usability of the interface. A comparative analysis of the application composition process with the previous visual interface was conducted to quantify the workload reduction realized by the new interface. Workload was measured as the number of user interactions (mouse or keyboard) required to compose an application. On average, application composition effort was reduced 44.0% for the test cases.

14. SUBJECT TERMS Man Computer Interface, Interfaces, Software Engineering, Computer Graphics			15. NUMBER OF PAGES 148
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL